CONTROL OF CONCURRENT OPERATIONS IN ASYNCHRONOUS DIGITAL PROCESSES

Fred U. Rosenberger

Washington University

Prepared for:

Advanced Research Projects Agency
Public Health Services

July 1970

DISCLAIMER NOTICE

THIS DOCUMENT IS THE BEST QUALITY AVAILABLE.

COPY FURNISHED CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

CONTROL OF CONCURRENT OPERATIONS IN ASYNCHRONOUS DIGITAL PROCESSES

Fred U. Rosenberger

TECHNICAL REPORT NO. 14

July, 1970

Computer Systems Laboratory
Washington University
St. Louis, Missouri

This research, under the direction of Dr. Donald F. Wann, was originally presented to the Electrical Engineering Department of Washington University as a doctoral dissertation. The work was supported in part by the Advanced Research Projects Agency of the Department of Defense under Contract SD-302 and by the Division of Research Facilities and Resources of the National Institute of Health under Grant FR-00396.

ABSTRACT

Methods are presented which can be used to analyze a sequential digital process and synthesize a process which performs the same operations but in less time by allowing concurrent execution of operations where possible. In the model used, concurrent execution of operations is controlled by branch operations which initiate concurrent paths of execution and by rendezvous operations which combine concurrent paths of execution after completion. Two types of errors which may occur in concurrent processes but not in sequential processes are distinguished. The first, called sequencing errors, cause the computed results to depend on the magnitude of the delays in the process operations and are due to operations being initiated before all of their data values are available. The second type of errors are called implementation errors and are caused by attempting to combine non-concurrent paths of execution with a rendezvous operation and by attempting to initiate an operation that is already being executed. The problem of detecting and correcting these errors is eliminated by insuring that the synthesized concurrent process is free of sequencing and implementation errors. A precedence relation is determined for the operations of the process and is used to insure that no operation is executed until all operations whose execution must precede it are completed thereby preventing sequencing errors. Dominance relations and directed cut sets, which specify the relationship between the execution of the operations, are used to avoid implementation errors.

TABLE OF CONTENTS

TABLE OF CONTENTS

(cont'd)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(cont'd)

LIST OF FIGURES
(cont'd)

CONTROL OF CONCURRENT OPERATIONS IN ASYNCHRONOUS DIGITAL PROCESSES

1. INTRODUCTION

Despite the tremendous increases in digital system operating speeds over the past twenty years, many applications require, or can profitably utilize, even faster systems. Many problems, such as numerical weather prediction, have restrictions on the time by which results must be produced if those results are to be useful. A prediction of 95% probability of rain for yesterday has very little value. Also, if digital systems can be made faster, without a corresponding increase in price, the cost per calculation or problem can be reduced assuming there is enough demand to utilize the increased capability. Although increased operating speed can be achieved by developing faster hardware elements such as logic gates and memories, there is a limit to the component operating speeds with the state of technology available at any given time. There is also a limit to the speed with which arithmetic operations such as addition and multiplication can be performed with a given set of logic elements.[1, 2, 3] This report will be concerned with increasing the operating speed of processes by changes in the organizational structure, specifically, changes that will allow operations to be executed concurrently. The goal of the research reported here was to develop methods for analyzing a process in which all operations are executed sequentially and to then synthesize an *error-free* process that executes the same operations as the original process, but in less time, by executing operations concurrently when possible.

The processes discussed will be represented by flow charts like the example in Figure 1, which represents a set of operations to be executed either by a programmed computer or by a special purpose digital system. The flow charts are composed of three types of operations: *processing operations*, denoted by rectangular boxes, that modify the data in storage elements (e.g., add, gate, write); *decision operations*, denoted by diamond shaped boxes, that monitor the data in storage elements and determine which following operations are to be performed (e.g., compare); and *control operations*, denoted by circles, that, together with the directed lines connecting the operations, determine the order in which operations are to be performed. The processing and decision operations are numbered in the lower right corner for reference. Storage elements and data paths are not shown explicitly by the flow chart but are implied by the operations. Each processing operation has an input terminal or connection called the *initiation terminal* and an output terminal or connection called the *completion terminal*. A signal applied to the initiation terminal of a processing operation is called an *initiation signal* and causes that operation to be executed. When the execution of the operation is completed it generates a signal at its completion terminal called the *completion signal*. If the completion terminal of one operation is connected to the initiation terminal of a second operation, the second operation will begin execution as soon as the execution of the first operation is completed.

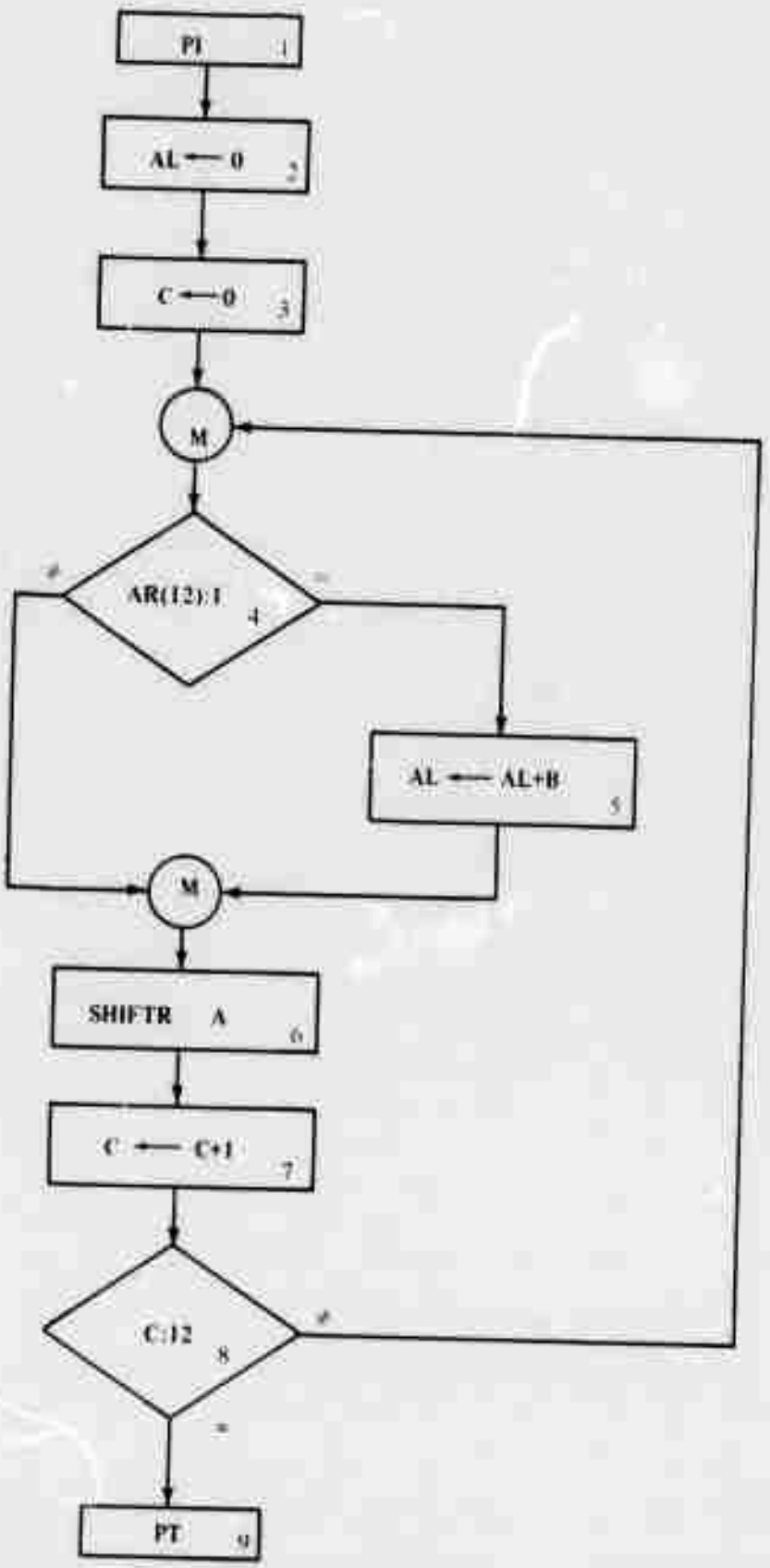


FIGURE 1. EXAMPLE OF A PROCESS

The decision operations have a single initiation terminal and two or more completion terminals. When a decision operation is executed it generates a completion signal on one of its completion terminals, the particular terminal depending on the decision outcome. The *branch* operation has one initiation terminal and two or more completion terminals. When it is executed it generates a signal on each of its outputs and this can be used to initiate concurrent operations.

The *merge* operation has two or more initiation terminals and a single completion terminal. When it receives an initiation signal on any one of its inputs it generates the completion signal. The *rendezvous* operation also has two or more initiation terminals and a single completion terminal but it generates a completion signal only after it has received an initiation signal on each of its inputs. The merge is used to combine the completion signals from operations where only one can be executed at a time while the rendezvous is used to combine the completion signals from operations that are executed concurrently. The first operation in a process is always a *process termination* or PT operation. Execution of a PI operation begins the execution of the corresponding process and the execution of the process continues until the PT operation for the process is initiated.

The processing and decision operations are not restricted to simple operations but may represent long calculations or subroutine calls that are grouped together as one operation for convenience. A flow chart with no branch and rendezvous operations is called the flow chart of a sequential process or a *sequential flow chart* while a flow chart with branch and rendezvous operations is a flow chart of a concurrent process or a *concurrent flow chart*. The execution time of each operation is assumed to be variable and not known although average execution times may often be estimated. We will assume that the execution times of the control operations are short compared to those for the processing and decision operations and can therefore be ignored. Since each operation produces a completion signal that is used to initiate following operations, the control of the process described does not depend on the execution time of the operations, and the systems are asynchronous.

As was mentioned previously the flow charts may represent a set of operations to be executed by a program in a digital computer. In that case the branch operation corresponds to the FORK instruction and the rendezvous operation corresponds to WAIT or JOIN instructions that have been proposed by other authors [4, 5, 6]. These instructions allow more flexibility in controlling concurrent operations that the DO TOGETHER or PARALLEL FOR[7, 8], which specify groups of operations to be executed concurrently. Figure 2 shows an example proposed by Fisher[9] of a concurrent process that cannot be specified by DO TOGETHER statements.

The flow charts can also represent a process to be executed by a special purpose digital system. Such a system can be constructed conveniently from the set of macromodules that are currently being developed in the Computer Systems Laboratory at Washington University[10, 11, 12, 13]. The examples of systems in the following discussion will be given in terms of macromodule operations.

Systems with concurrent operations are subject to two types of errors that do not occur in sequential systems. The first type, called a *sequencing error*[14], occurs when some computed value depends on the relative timing between concurrent operations. A simple example of this would be the case where concurrent operations added a number to, and tested the same register. There would be no way to predict whether the register contained the original value, the sum, or something in between when the test was made. Such an error can be detected but cannot be corrected without determining whether the addition should be performed before or after the register test. The second type of error, which is called an *implementation error*[14], is caused by incorrect use of branch and rendezvous operations. An example of this type of error is shown in Figure 3 where the rendezvous operation will receive only

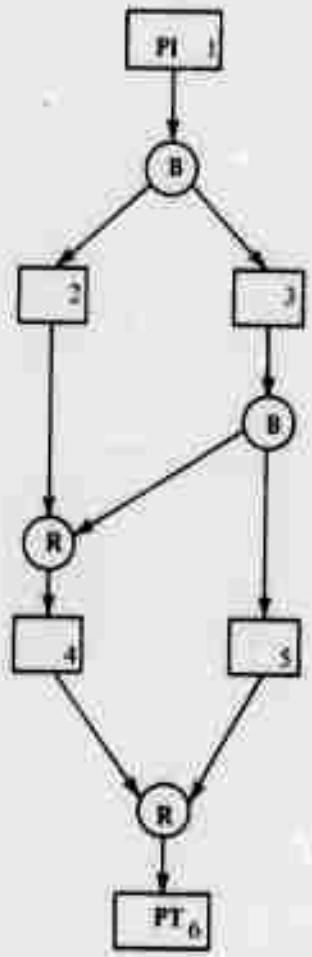


FIGURE 2. EXAMPLE OF A CONCURRENT PROCESS THAT CANNOT BE SPECIFIED BY DO TOGETHER STATEMENTS

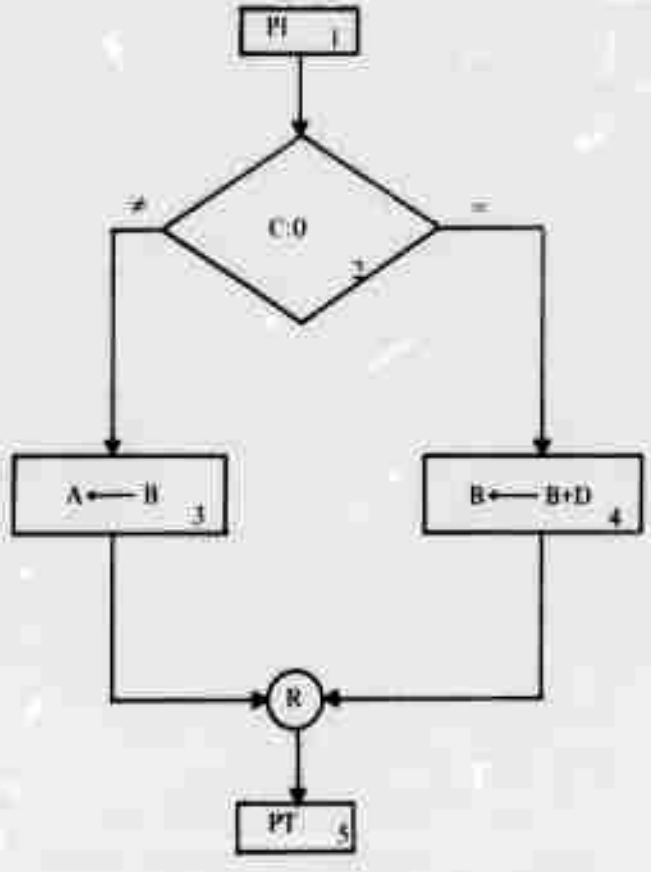


FIGURE 3. INCORRECT USE OF RENDEZVOUS OPERATION

one input signal. In this example the PT operation will never be initiated. Several other types of implementation errors are possible and these will be discussed later.

Several authors[15, 16, 17, 18] have discussed the problem of assigning computers or processors to the operations of a process. Estrin and Turn, and Martin and Estrin have discussed the case of the restructurable computer where the "cost" of executing a particular operation depends on which processor executes it. They describe iterative methods by which trial assignments of each operation to a processor are made and the total computation cost determined. The assignments are then perturbed and the new cost compared with the preceding one to determine whether an improvement has been made. Schwartz[19] has discussed a heuristic method for assigning processors to a set of operations. A different model of a process is used by Karp and Miller[20] and by Reiter[21] in which the connections between operations represent queues of data. An operation may be executed whenever each of its inputs contains a data value and upon completion of the operation a data value is placed on some of the output lines. Data values are allowed to queue between the operations and Karp and Miller, and Reiter have investigated bounds on the length of these queues and the necessary conditions for a process to terminate.

Detection of errors in concurrent systems has been discussed in several papers[14, 22, 23]. A summary of methods for detecting implementation errors is given by Keller and Wann[23]. These include topological analysis, simulation, and the state transition method which is a systematic method of testing each possible state of control signals in a process. Van Horn[24, 25] has considered the problems of insuring that a concurrent process executed on a digital computer will always produce the same results if started from the same initial state, and he discusses a number of restrictions which must be satisfied. These restrictions do not eliminate errors that may occur in the process but insure that the results will be reproducible so that debugging will be facilitated.

In contrast to the preceding work on detecting errors or insuring consistent performance of processes with errors, the work reported here is concerned with synthesizing concurrent processes that are free of errors. The work of Bernstein[26] and Fisher[9] in determining precedence relations, which are used to specify the ordering of operation executions required to prevent sequencing errors, is reviewed and several other relations useful in the synthesis of concurrent processes are developed in Chapter 2. These relations are used in the synthesis of error-free concurrent processes for loop-free sequential processes and for sequential processes containing loops, in Chapters 3 and 4 respectively. Chapter 5 summarizes the results, lists some deficiencies in the approach, and makes suggestions for further work.

2. CONNECTIVITY, PRECEDENCE, AND DOMINANCE RELATIONS

This section discusses the use of binary relations, which may be represented by ordered pairs, directed graphs, or boolean matrices to describe several relations between the operations of a process. The connectivity relation describes the order of execution of operations in a sequential process, the precedence relation describes the order of execution that must be maintained to prevent sequencing errors, and the dominance relation describes the relationship between the execution of operations.

2.1 BINARY RELATIONS

A binary relation on a set of elements, $X = \{x_1, x_2 \ldots x_n\}$, can be specified by a set of ordered pairs of elements of X. For example, the relation of "less than" on the set of integers from 1 to 4 can be specified by the following set of pairs of numbers:

$$\{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$$

This set contains all pairs, (x_i,x_j), of elements of X such that x_i is less than x_j. For a set with n elements there are n^2 possible ordered pairs of elements and each may be included in the relation or not. Therefore, there are $2^{(n^2)}$ possible binary relations on a set of n elements.

There are several important properties that binary relations *may* possess. The first of these is that the relation may be *transitive*. That is, if (x_i,x_j) and (x_j,x_k) are members of the relation, (x_i,x_k) must be a member also. The relation "less than" in the previous example is a transitive relation. In this example the relation can be specified by the ordered pairs (1, 2), (2, 3), and (3, 4) and the fact that the relation is transitive, since the other members of the relation can be derived from this.

A second property of binary relations is that they may be *symmetric*. That is, if (x_i,x_j) belongs to the relation then (x_j,x_i) must belong to the relation also. A relation is *antisymmetric* if (x_i,x_j) being a member of the relation implies that (x_j, x_i) cannot be a member of the relation. A third property of binary relations is that they may be reflexive or antireflexive. A relation is *reflexive* if for all x_i, (x_i,x_i) is a member of the relation and *antireflexive* if for all x_i, (x_i,x_i) is not in the relation.

The members of a relation are *topologically ordered* if they are numbered so that if (x_i,x_j) is a member of the relation then $i < j$. A relation can only be topologically ordered if it is antisymmetric and antireflexive.

In addition to being represented by pairs of elements, a binary relation can also be represented by a directed graph or by a square boolean matrix. In the directed graph representation, each element of the set, X, is represented by a node or vertex of a graph and each member, (x_i,x_j), of the relation is represented by an arc from node x_i to node x_j. In the boolean matrix representation, each element of the set is represented by a row and column of the matrix and each member of the relation, (x_i,x_j) is specified by a '1' in the i, j position of the matrix. If (x_i,x_j) is not a member of the relation then the i, j position of the matrix is a '0'. If the row and column numbers of a matrix represent a topological order for a relation the matrix will be superdiagonal. The directed graph and boolean matrix representations of the "less than" relation for integers 1 to 4 are given in Figure 4.

Each of the three representations of binary relations discussed has certain advantages. The directed graph is easiest for humans to visualize and follow but cannot be used directly for digital computer calculations; the boolean matrix representation is convenient for computer calculations using logical operations; the ordered pair representation can be manipulated using list processing operations [27, 28, 29]. If the relation contains many members, the boolean matrix representation will usually require less computer storage than the ordered pair representation while if the relation has only a few members the converse is usually true. In the remainder of this report, the examples will be illustrated by directed graphs and methods for calculating properties of relations will be discussed in terms of boolean matrices.

The operations of a process will be represented by the nodes of directed graphs and we will use "operation" and "node" interchangeably. The first node of a graph will always represent the PI operation and the last node will always represent the PT operation for a process.

2.2 OPERATIONS ON BINARY RELATIONS

2.2.1 Transitive Closure

The first operation to be discussed is the determination of the transitive closure of a transitive binary relation. By *transitive closure* is meant all members of the relation, including those implied by the given members but not specified explicitly. A relation that is equal to its transitive closure will be called a *complete relation* and the corresponding boolean matrix and directed graph will be called a *complete matrix* and *complete graph*, respectively. The members implied by two or more given members can be found by taking powers of the boolean matrix, A, which represents the relation as shown by Prosser[30]. The expression for the i,j term of the square of the matrix is given by the boolean expression:

$$(A^2)_{ij} = \sum_{k=1}^{N} A_{ik} A_{kj}$$

Thus, the i,j term of the squared matrix is equal to '1' if and only if there is some k such that $A_{ik} = A_{kj} = 1$. Then every '1' in A^2 represents a member of the relation which is implied by two other members. Similarly, members of the relation which are implied by a chain of n members will be represented by a '1' in the nth power of the matrix. Every member of the relation must be implied by a chain of members of length N − 1 or less where N is the number of elements in the set, X. Therefore, the complete set of members of the relation, including all implied relations, can be found by adding together (modulo 2) all of the powers of A from 1 to N − 1. The computation required can be reduced by noting that [30]:

$$(A^2+A)^2 = A^4+A^3+A^2 \text{ and in general that}$$

$$\underbrace{(\ldots((A^2+A)^2+A)^2\ldots)^2}_{n \text{ times}} = \sum_{i=2}^{2^n} A^i$$

Unclassified
Security Classification

AD-771743

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*

Computer Systems Laboratory
Washington University
St. Louis, Missouri

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3. REPORT TITLE

Control of Concurrent Operations in Asynchronous Digital Processes

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Interim

5. AUTHOR(S) *(First name, middle initial, last name)*

Fred U. Rosenberger

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
July, 1970	108	54

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
(1) DOD(ARPA) Contract SD-302 (2) NIH(DRFR) Grant No. RR-00396	Technical Report No. 14
b. PROJECT NO. (1) ARPA Project Code No. 655	
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10. DISTRIBUTION STATEMENT

Distribution of this document is unlimited

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	ARPA – Information Processing Techniques, Washington, D.C., NIH., Div. of Research

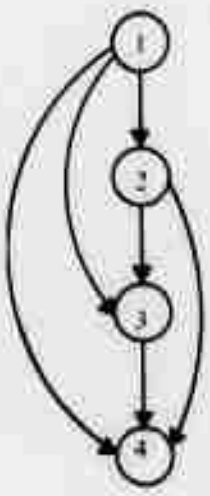
13. ABSTRACT

Methods are presented which can be used to analyze a sequential digital process and synthesize a process which performs the same operations but in less time by allowing concurrent execution of operations where possible. In the model used, concurrent execution of operations is controlled by branch operations which initiate concurrent paths of execution and by rendezvous operations which combine concurrent paths of execution after completion. Two types of errors which may occur in concurrent processes but not in sequential processes are distinguished. The first, called sequencing errors, cause the computed results to depend on the magnitude of the delays in the process operations and are due to operations being initiated before all of their data values are available. The second type of errors are called implementation errors and are caused by attempting to combine non-concurrent paths of execution with a rendezvous operation and by attempting to initiate an operation that is already being executed. The problem of detecting and correcting these errors is eliminated by insuring that the synthesized concurrent process is free of sequencing and implementation errors. A precedence relation is determined for the operations of the process and is used to insure that no operation is executed until all operations whose execution must precede it are completed thereby preventing sequencing errors. Dominance relations and directed cut sets, which specify the relationship between the execution of the operations, are used to avoid implementation errors.

DD FORM 1 NOV 68 1473 REPLACES DD FORM 1473, 1 JAN 64, WHICH IS OBSOLETE FOR ARMY USE.

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Parallel Processing						
Concurrent Processing						
Directed Graphs						
Dominance Relation						
Precedence Relation						



$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

FIGURE 4. REPRESENTATION OF THE RELATION LESS THAN

Thus, only $\lceil \log_2 (N-1) \rceil$ boolean multiplications are required to find all implied members of the relation where $\lceil a \rceil$ indicates the smallest integer greater than or equal to a.

A more efficient method has been described by Warshall[31] and he gives a proof that his method is equivalent to calculating the powers of A. The matrix is scanned one column at a time and when a '1' is found in position i,j, row i is replaced by the sum (modulo 2) of row i and row j as shown below:

```
for j = 1, 2 . . . N
    for i = 1, 2, . . . N
        if A(i, j) = '1'
            for k = 1, 2, . . . N
                A(i, j) = A(i, k) + A(j, k)
```

This algorithm operates only on rows of the matrix and can be conveniently implemented in a digital computer if the matrix is stored by rows.

If the A matrix is superdiagonal, the following algorithm can be used:

```
for i = N, N-1 . . . 1
    for j = N, N-1 . . . i+1
        if A(i, j) = '1'
            for k = i, i+1 . . . N
                A(i, k) = A(i, k) + A(j, k)
```

2.2.2 The Cover Of A Relation

By the cover of a transitive relation is meant the minimum set of members of the relation that can be used to specify it. This is formed by deleting all members of the relation that are implied by two or more other members. A relation that is equal to its cover will be called a *basic relation* and the corresponding boolean matrix and directed graph will be called a *basic matrix* and *basic graph*, respectively. The members of the relation that are implied by other members can be found by taking powers of A as described in Section 2.2.1 or by calculating the transitive closure of the sum of the second and third powers of A. The second power of A contains all members of the relation that are implied by two members and the third power of A contains all members of the relation that are implied by three members. The transitive closure of the sum of the second and third powers contains all members that are implied by combinations of two and three members or all members of the relation implied by two or more members. Deleting all of these implied members from the original A matrix will leave the basic relation or the cover for the relation.

As with the calculation of the transitive closure, a simpler method[32] can be used if the matrix is superdiagonal. The algorithm, as presented below, can be applied only to a complete relation and will not work if all of the implied relations are not included.

```
for i = 1, 2 . . . N - 1
    for j = i+1, i+2 . . . N
        if A(i, j) = '1'
            for k = i, i+1 . . . N
                A(i, k) = A(i, k) & NOT A(j, k)
```

2.2.3 Topological Ordering

A topological order for the elements of a set under a binary relation may be determined from the transitive closure or complete matrix representation of the relation as shown by Chen and Wing[33]. Of course if any entry on the main diagonal of the complete matrix is a '1', the relation is not antisymmetric and antireflexive so no topological ordering is possible. The elements are ranked according to the number of '1's in their row of the complete matrix with the element having the most '1's placed first. If two elements have the same number of '1's in their rows of the complete matrix they may be ranked in either order.

2.3 CONNECTIVITY AND REACHABILITY RELATIONS

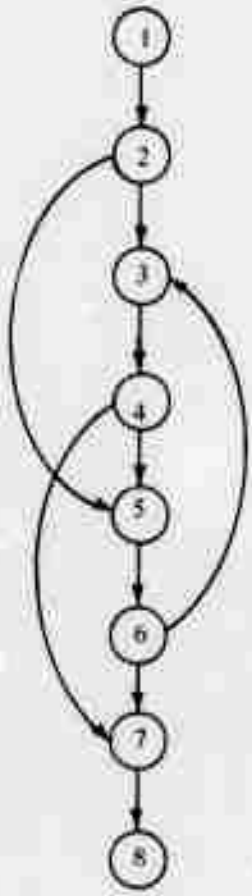
A *connectivity relation* is a binary relation that describes the order in which a set of operations that comprise a sequential process are to be performed. That is, if operation J can be performed directly following operation I then (I,J) is a member of the relation and there is an arc from node I to node J in the corresponding directed graph. The connectivity relation is not a transitive relation since J directly following I, and K directly following J does not imply that K directly follows I. The directed graph and boolean matrix that represent the connectivity relation will be called the *connectivity graph* and *connectivity matrix*, respectively. If the order of execution of the operations in the process is specified by a flow chart, the connectivity graph will be isomorphic to the flow chart and can be determined directly from it.

The *reachability relation* is the transitive closure of the connectivity relation. If the I,J term of the reachability matrix, R, is '1' then operation J may follow operation I but not necessarily directly. If J may follow I then I *reaches* J. Ramamoorthy[34, 35] has shown several uses of the connectivity and reachability relations. One of these, which will be of use in the following discussion, is a method to find the feedback arcs of a connectivity graph, or the members of the relation which must be deleted to make the relation antisymmetric and antireflexive (loop-free).

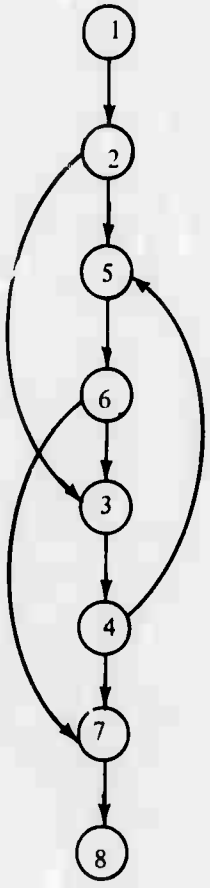
The feedback arcs of a connectivity graph are not necessarily unique as the examples in Figure 5 show. In Figure 5A the arc from node 6 to node 3 appears to be the feedback arc while in Figure 5B the feedback arc appears to be the one from node 4 to node 5. Actually the graph in Figure 5B is isomorphic to the one in Figure 5A and is redrawn in Figure 5C to show the symmetry. Either the arc from node 6 to node 3 or the arc from node 4 to node 5 can be considered the feedback arc for this graph. When either arc is deleted the graph becomes loop-free.

The feedback arcs of a connectivity graph are defined as the minimum set of arcs (not necessarily unique) which can be removed to eliminate all loops of the graph, and still allow all of the nodes to reach the termination node[35]. The restriction requiring all nodes to be able to reach the termination node after deletion of the feedback arcs is included to prevent selection of the arc from node 2 to node 3 in Figure 6 as the feedback arc. Alternatively we could require that all nodes be reachable from the initiation node after deletion of the feedback arcs, but the requirement chosen is more convenient for alter work. Both requirements can be satisfied for most connectivity graphs but not for the one shown in Figure 6. Either the arc from node 3 to node 4 or the arc from node 4 to node 2 can be selected as the feedback arc and when this is deleted either node 4 cannot reach the termination node or the initiation node cannot reach node 4.

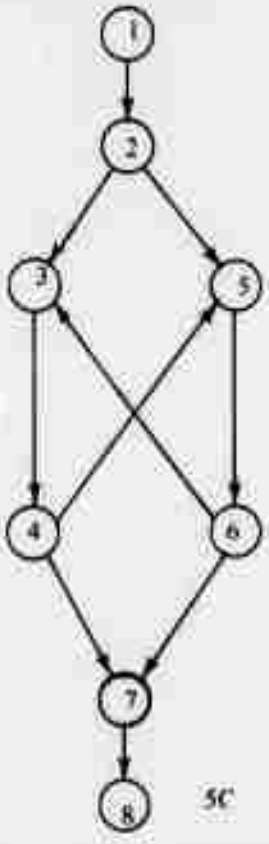
A method for determining the feedback arcs is to calculate the maximally strongly connected subgraphs (M.S.C. subgraphs) as suggested by Ramamoorthy[35]. The *M.S.C. subgraphs* are the maximum sets of nodes such that each node in the set can reach any other node in the set. If a graph has no M.S.C. subgraphs it has no feedback arcs. The M.S.C. subgraphs are clearly disjoint since by definition if two M.S.C. subgraphs have a node in common they must have all nodes in either M.S.C. subgraph in common. The M.S.C. subgraphs are found from M, which is formed by 'and'ing the reachability matrix and its transpose together. The I,J element of M is a '1' if and only if I can reach and be reached by J. The M.S.C. subgraphs of the system correspond to the distinct rows of M and are comprised of the nodes represented by the row positions containing a '1'. A node J, is an *entrance node* of an M.S.C. subgraph if it is a member of the subgraph and there is any node, I, which is not a member of the subgraph such that C(I,J) = '1' where C is the connectivity matrix. Similarly, a node, J, is an exit node if there is any node, K, which is not a member of



5A



5B



5C

FIGURE 5. A PROCESS WITH NO UNIQUE FEEDBACK ARC

FIGURE 6. A GRAPH IN WHICH ALL NODES CANNOT BE REACHED FROM THE INITIATION NODE AFTER DELETION OF THE FEEDBACK ARC

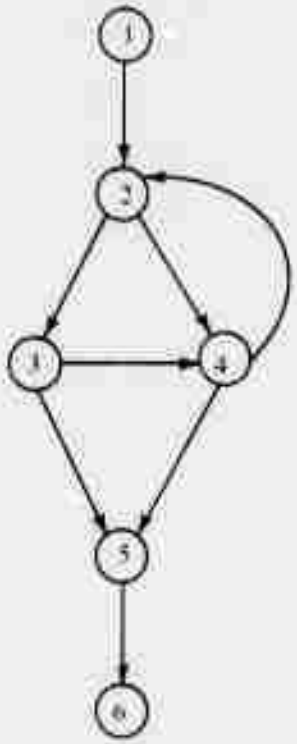


FIGURE 7. A GRAPH CONTAINING A STRONGLY CONNECTED SUBGRAPH WITH 2 EXIT NODES

the subgraph such that C(J,K) = '1'. Ramamoorthy identifies the feedback arcs as the arcs from nodes within each M.S.C. subgraph to the entrance node of the M.S.C. subgraph, but since we require each operation to be able to reach the termination node after the feedback arcs are deleted, we shall identify the feedback arcs as the arcs from exit nodes of a M.S.C. subgraph to a node within the M.S.C. subgraph. After the feedback arcs are determined, they will be deleted from the graph and the M.S.C. subgraphs of the remaining graph calculated to test for additional feedback arcs. If there is more than one exit node from a M.S.C. subgraph then the arcs from all of the exit nodes may not be feedback arcs as illustrated in Figure 7. The M.S.C. subgraph is composed of nodes 2, 3, and 4 and both nodes 3 and 4 are exit nodes. To obtain the fewest number of feedback arcs, the exit nodes can be selected one at a time, their output arcs deleted, and the feedback arcs in the remaining graph determined. The minimum set of arcs found is selected as the set of feedback arcs. In the example of Figure 7, selecting node 3 first and then node 4 gives two feedback arcs while selecting node 4 first gives only one feedback arc. Therefore, the arc from node 4 to node 2 is the feedback arc.

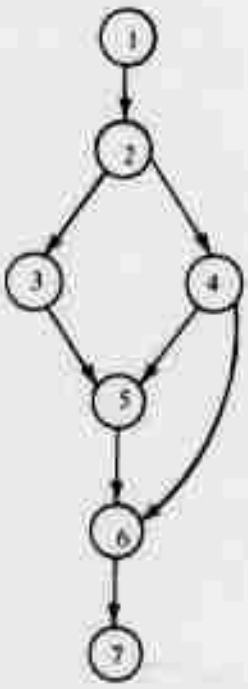
2.4 PRECEDENCE RELATIONS

A precedence relation on a set of operations specifies which operations must precede other operations to insure correct performance of the process. A precedence relation is transitive, antireflexive, and antisymmetric. The directed graph and boolean matrix that represent the basic set of members of the precedence relation will be called the *basic precedence graph* and *basic precedence matrix*, respectively, while the directed graph and boolean matrix representing the complete set of members of the precedence relation will be called the *complete precedence graph* and *complete precedence matrix*, respectively. Operations which must be completed before a given operation will be called the *predecessors* of the operation. The predecessors of operation I are all operations, J, with a '1' in the Jth row of the Ith column of the complete precedence matrix. The *direct predecessors* of I are the predecessors of I that are not predecessors of any other predecessor of I. These correspond to members of the basic precedence relation and are all operations with a '1' in their row of the Ith column of the basic precedence matrix. The *indirect predecessors* are the predecessors of I which are not direct predecessors of I. Most of the following work will be concerned with the basic precedence relation since if all of its members are satisfied all of the members of the complete precedence relation are satisfied also.

2.4.1 Precedence Relations Required by Decision Dependence

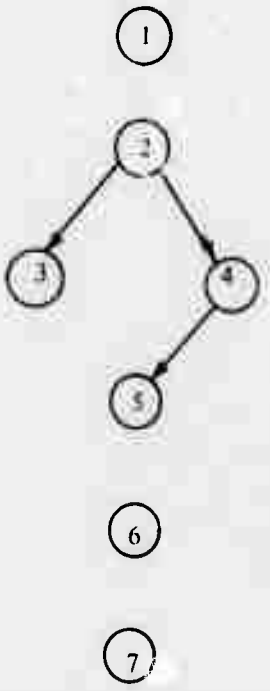
An operation may not be executed unless it would also be executed in the sequential process. We do not consider the possibility of executing operations before it can be determined that they are to be executed. Thus, each operation may be executed only after the completion of all decisions that can prevent the execution of that operation[9].

If a decision, I, can prevent the execution of an operation, J, there must be some outcome of decision I for which operation J may be executed and some outcome of decision I for which operation J cannot be executed. A particular outcome of decision I is represented by an arc of the connectivity graph from node I to some other node K. These operations may be found from the reachability matrix. The union of the sets of operations that may be



A CONNECTIVITY GRAPH

FIGURE 8



THE DECISION DEPENDENT PRECEDENCE RELATION FOR THE CONNECTIVITY GRAPH OF FIGURE 8

FIGURE 9.

executed for each decision outcome minus their intersection gives the set of operations that can be executed for only some of the decision outcomes, or all operations that the decision can prevent from being executed. Thus, the operations that must be executed after each decision can be determined by manipulations on the reachability matrix. Figures 8 and 9 show a connectivity graph and the members of the precedence relation due to decision dependence for this graph. Operations 3, 4, and 7 can be reached by the output from decision 2 and operations 3, 5, 6, and 7 can be reached by the other output from decision operation 2. The union of these two sets minus their intersection gives operations 3 and 4 which must follow operation 2. Similarly the sets of operations reached by the two outcomes of decision operation 4 are $\{5, 6\}$ and $\{6\}$ and their union minus their intersection is just $\{5\}$. Therefore, operation 5 must follow operation 4 as shown in the decision dependent precedence relation in Figure 9.

2.4.2 Precedence Relations Required by Data Dependence

The second class of precedence relations to be examined is due to storage elements whose values are changed by the operations[9, 26]. In the following discussion, *register* will be used as a general term to refer to any storage element whether it is a register, core memory, or other type. *Read* will be used to indicate any use of the contents of a register and *write* will refer to any change of the contents of a register. The types of operations considered are decision operations which can affect the flow of control based on the contents of registers, and processing operations which change the contents of a register based on the contents of other registers. It is assumed that the actual registers used can be determined by analysis of the operations of the process, which eliminates or restricts the possibility of using indirect addressing. This is a serious limitation but one which apparently cannot be avoided. The original order between two operations must be maintained if their order of execution determines what value is left in some register when they are completed, or if their order of execution can affect the value in a register read by one of the operations. Consider two operations, I and J, with I preceding J. Clearly, if they both write the same register then their order of execution cannot be interchanged since this would leave the value written by operation I in the register instead of that written by operation J. Therefore, if two operations write the same register, their order may not be interchanged.

If a register written by operation I is read by operation J, then the operations cannot be interchanged since, if they were interchanged the value read by operation J would not be the one written by operation I as in the original order of execution. Similarly, if operation J writes a register that is read by register I, the operations cannot be interchanged. Let R(X) represent the set of registers read by operation X, W(X) represent the set of registers written by operation X, and ϕ be the null set. Then if the following three conditions are satisfied, operations I and J *can* be executed in either order or concurrently.

1. $W(I) \cap W(J) = \phi$
2. $W(I) \cap R(J) = \phi$
3. $R(I) \cap W(J) = \phi$

Conversely, if any of the three relations is not true, the original order of the operations must be maintained.

The preceding discussion assumes there are no operations between I and J. If there are other operations between them, J may be required to follow I due to implied members of the precedence relation. It is interesting to note that

if the original ordering was operation I followed by operation J, then only the second relation above actually forces the operations to be sequential (e.g., the case where operation J requires results computed by operation I). In the first and third relations the ordering is constrained because both operations use the same register. If a different register were provided for operation J to write into, then the operations could be executed concurrently. This possibility will not be investigated further in this study since it would require examination of the register and modification of the operations specified originally.

A sequence of operations that is called as a subroutine can be considered as a single operation that is substituted for the call. The registers read and written would be the union of all those read and written by the operations of the sequence. Then the operation that was substituted for the call can be compared with the other operations to determine the required members of the precedence relation. In some cases it may be possible for two sequences to be done in either order but not concurrently. These are called commutative sequences by Bernstein[26]. This occurs when there is a register that both sequences write before reading and whose contents are not required by other operations outside of the sequences. In this case both sequences are using it as a scratch register and its contents are not required by other operations outside of the sequences. Detection of this situation requires recording not only which registers are read and written but also the order in which they are read and written and determining whether the registers are read by any following operations[26]. Figure 10A shows an example of possible concurrent operation that will not be detected by the method described above. In this example the sequences 2 to 4 and 5 to 7 can be executed in either order but not concurrently. However, if sequence 5 to 7 is executed first, then operation 8 can be executed concurrently with sequence 2 to 4, while executing sequence 2 to 4 first allows no concurrent operation. If a new register, D, is substituted for register A in operations 5 to 7 then the two sequences can be executed concurrently as shown in Figure 10B. In general, sequences that can be executed in either order but not concurrently, because of common use of a register for temporary storage, can be executed concurrently if additional registers are provided.

2.4.3 Combination of Precedence Relations

Combining the precedence relation due to decision dependence with that due to data dependence gives the precedence relation for the operations of a process. In addition we require every operation to follow the process initiation operation and to precede the process termination operation. In general the combination of these precedence relations will not be the basic or the complete precedence relation but the techniques discussed in Section 2.2 can be used to find the basic and complete precedence relations.

Another method, which calculates the basic precedence relation and complete precedence relation directly, is an algorithm due to Bingham, Fisher, and Semon[9, 36, 37] which is shown in Figure 11. Initially T is the matrix of precedence relations due to the decision operations and S is the union of the connectivity matrix and T. The algorithm converts these respectively into the complete and basic precedence matrices. The minimum number of comparisons of registers read and written are made by using relations already established. For instance, if there are three operations, where 2 must follow 1 and 3 must follow 2, there is no need to compare the registers read and written by 1 and 3 since there is already an implied precedence relation between them.

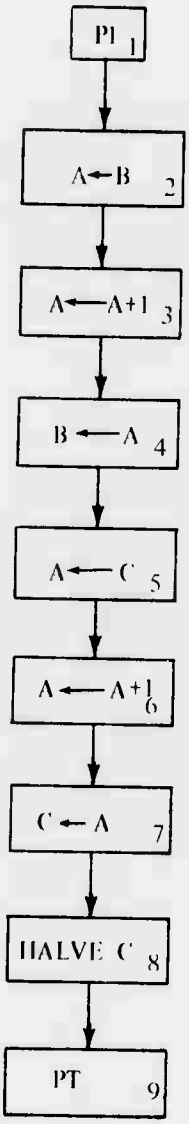


FIGURE 10A
A PROCESS WITH COMMUTATIVE
SEQUENCES

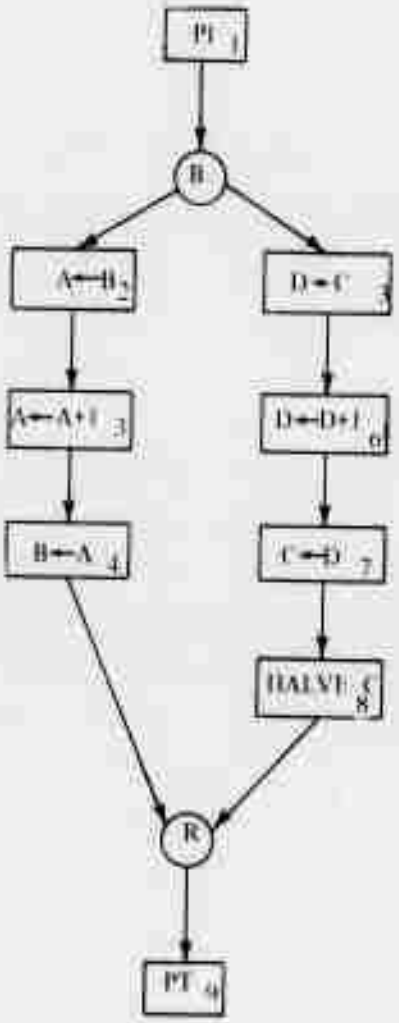


FIGURE 10B
THE PROCESS OF FIGURE 10A
WITH AN ADDITIONAL REGISTER

FIGURE 10. EXAMPLE OF THE USE OF AN ADDITIONAL REGISTER TO REMOVE PRECEDENCE REQUIREMENTS

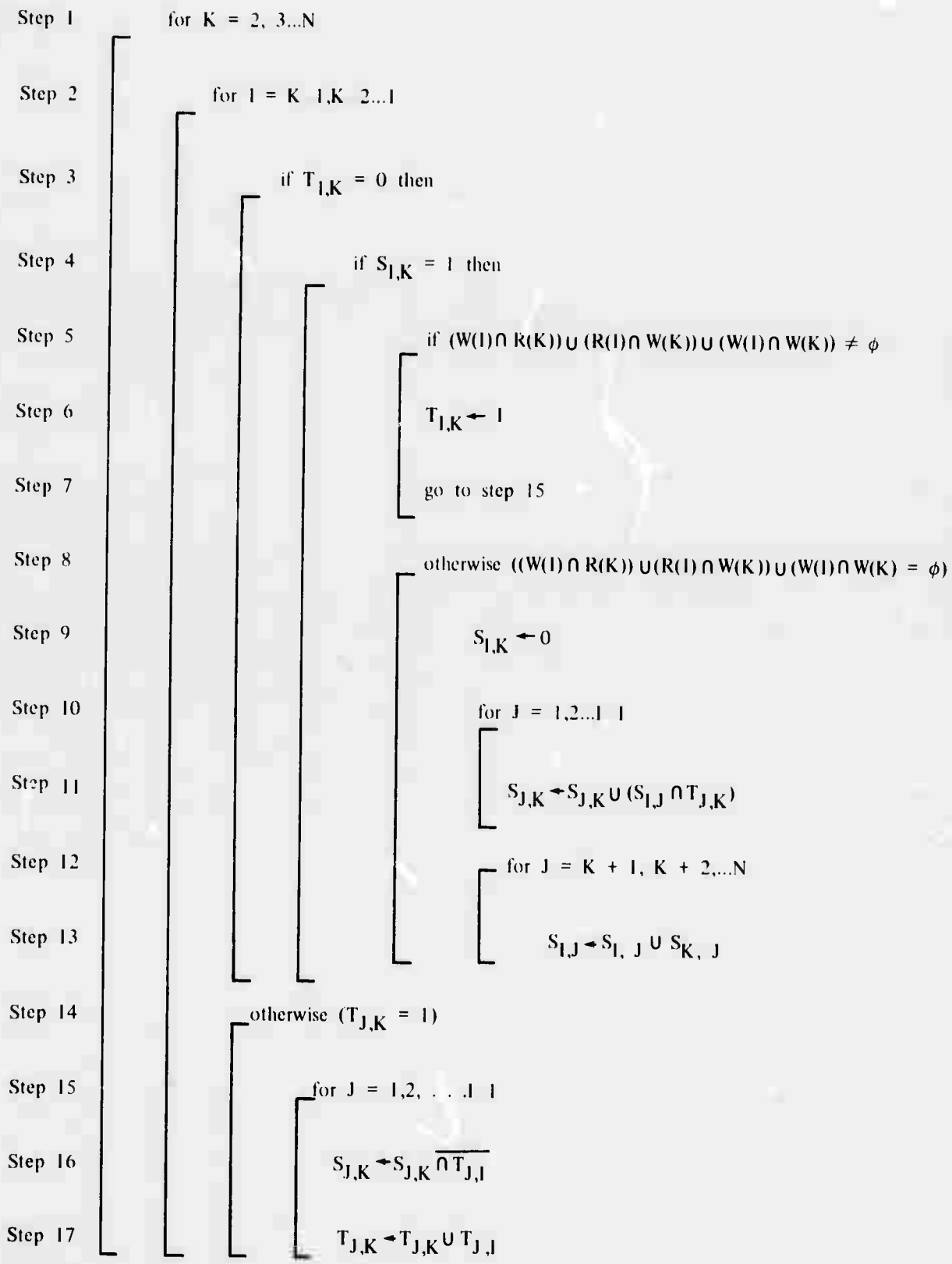


FIGURE 11. ALGORITHM TO CALCULATE THE PRECEDENCE RELATION

2.4.4 Limitations in Determining the Precedence Relation

There are several types of possible concurrent operation which are not recognized by the methods described above. A degenerate case can occur when a particular outcome of a decision can never be taken. An example is shown in Figure 12 where the "YES" outcome from decision 4 can never be taken. Since only the "NO" outcome from decision 4 can be taken, decision 4 and operation 5 could be deleted from the process, allowing operations 3 and 6 to be executed concurrently. In some cases it might be possible to detect decision outcomes that cannot occur but their occurrence is probably rare, and Bernstein[26] has shown that it is impossible to detect all such cases.

Another situation with possible concurrent operations which will not be recognized by the methods described is shown in Figure 13, where operations 3 and 4 are identical. In this case the shift operation is executed for either output of the decision but the techniques used for analysis will make it follow the decision. The concurrent execution of the shift and decision operations would be recognized if the shift operation was placed before the decision and the repetition of the operation would be eliminated. To detect this possibility would require comparing the operations executed for each outcome of a decision to determine whether the same operation is executed for each decision outcome[18].

A more serious limitation, which was mentioned earlier, is that some operations cannot be executed concurrently because they require use of the same register or storage element although there is nothing inherent in the data or operations to require them to be executed sequentially. These situations can be recognized but require changes in the operations specified and may require additional processing operations and registers to allow concurrent execution of operations. Another possibility would be to remove the member of the precedence relation between them but to interlock their actual execution[11] so that only one of them could be executed at a time. This would allow the first one whose predecessors were completed to be executed first but would prevent them from being executed concurrently. Another situation where operations may be executed in either order but not concurrently occurs when several operations test and alter common status information as described by Dijkstra[39]. Again, if this sharing of data can be recognized, the operations may be interlocked so that they may be executed in any order but only one can be executed at a time.

Finally, the operations of a process may be required to be executed sequentially when there is an alternate set of operations that will allow some concurrent execution. The value of the polynomial $X^5 + X^3 + X$ can be calculated by either of the methods shown in Figure 14, where the value computed by each operation of the flow chart is shown to the left of the operation. Each method requires four multiplications and two additions but several steps can be performed concurrently using the second method. This probably represents the most serious problem in allowing concurrent execution of operations although some work has been done on constructing algorithms for particular problems which allow concurrent execution of operations[40-46].

2.5 DOMINANCE RELATIONS

The dominance relation provides information on the relationship between the execution of the operations in a process. For example, if it is known that operation 5 in the example shown in Figure 15 is executed for a particular initiation of the process, operations 1, 2, and 7 must be executed also, operation 6 cannot be executed, and either

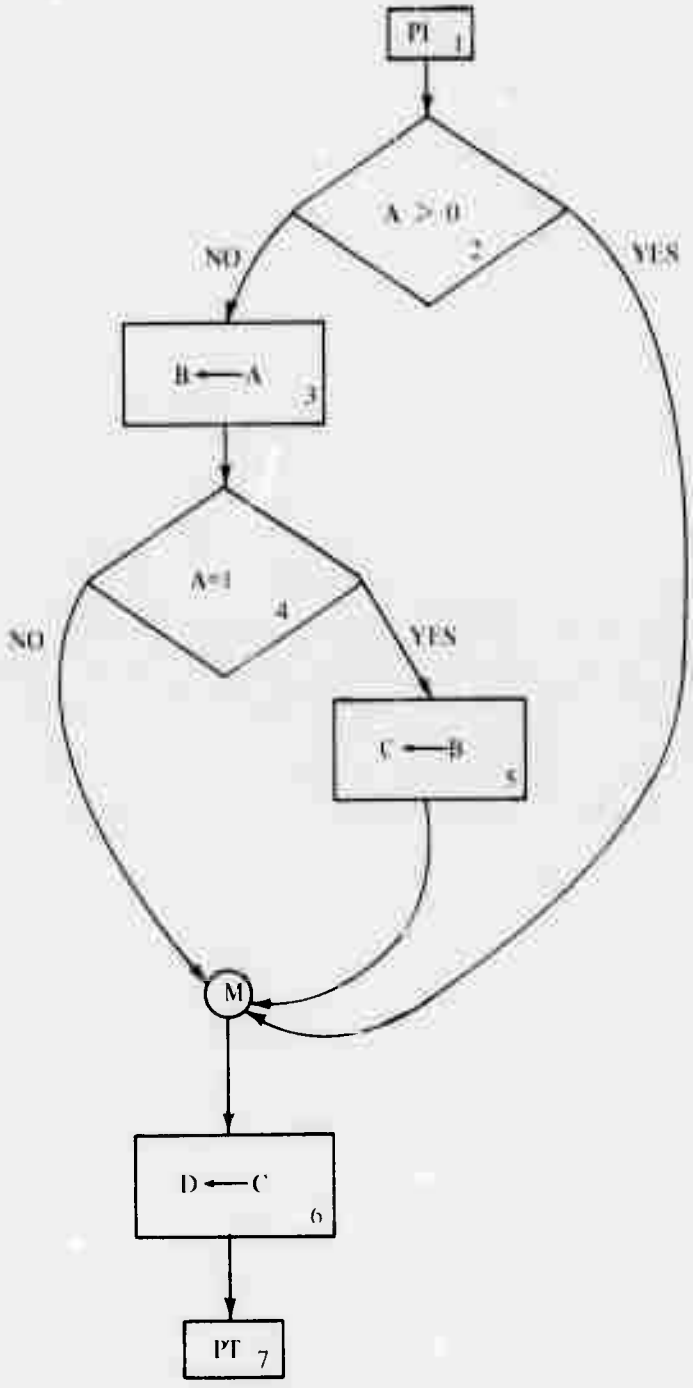


FIGURE 12. A PROCESS WITH A DECISION OUTCOME THAT CANNOT OCCUR

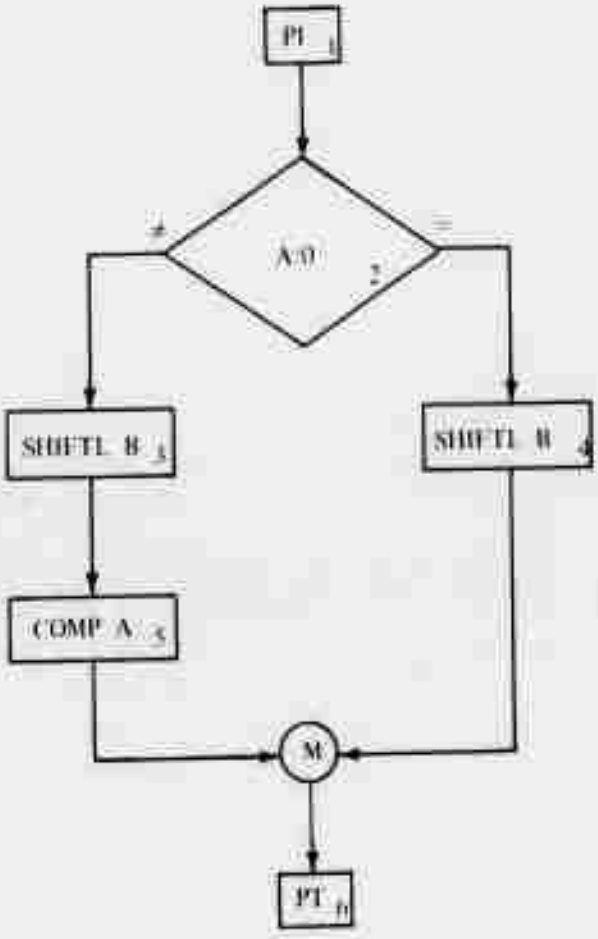
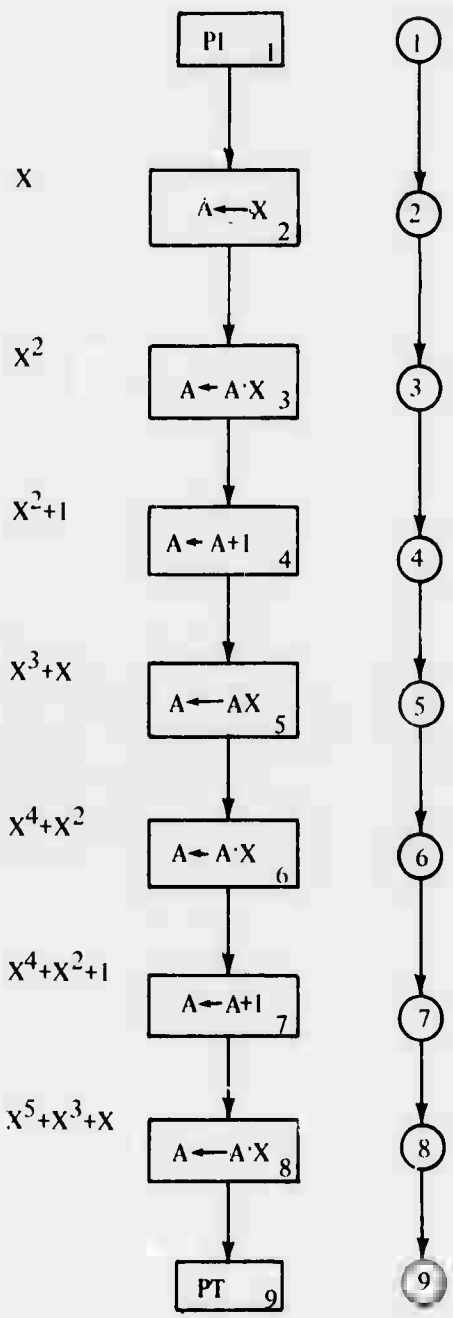


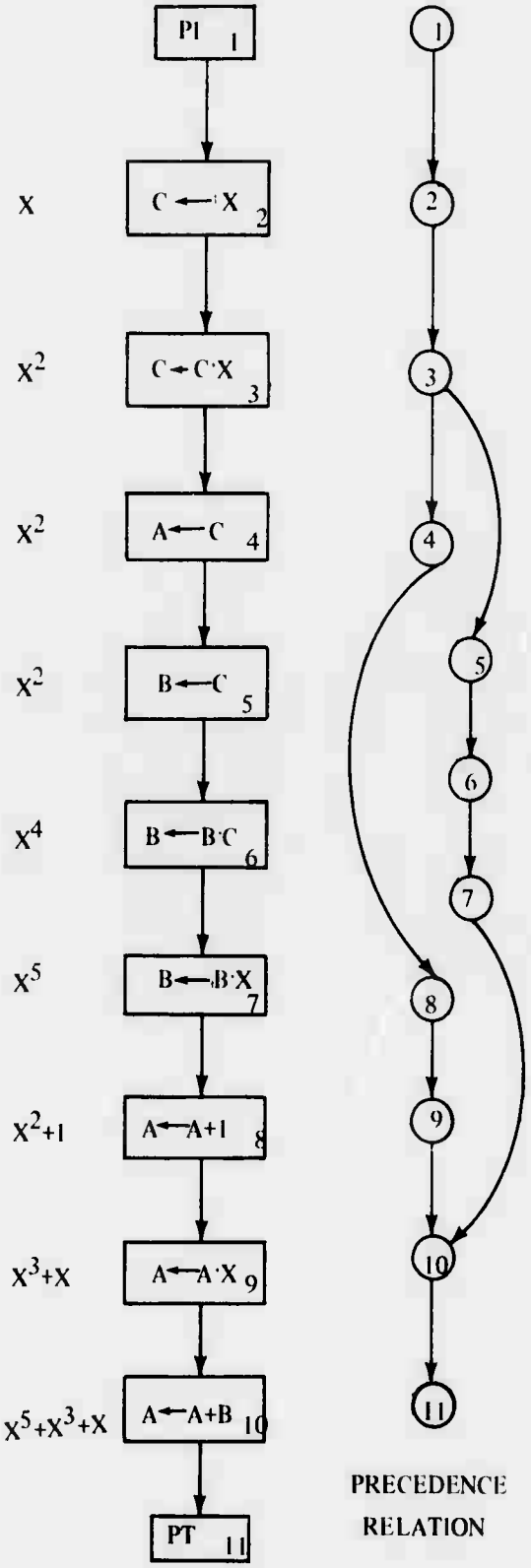
FIGURE 13. EXAMPLE OF A SEQUENTIAL PROCESS WITH THE SAME OPERATION EXECUTED FOR EACH OUTCOME OF THE DECISION



FLOW CHART

PRECEDENCE RELATION

METHOD 1

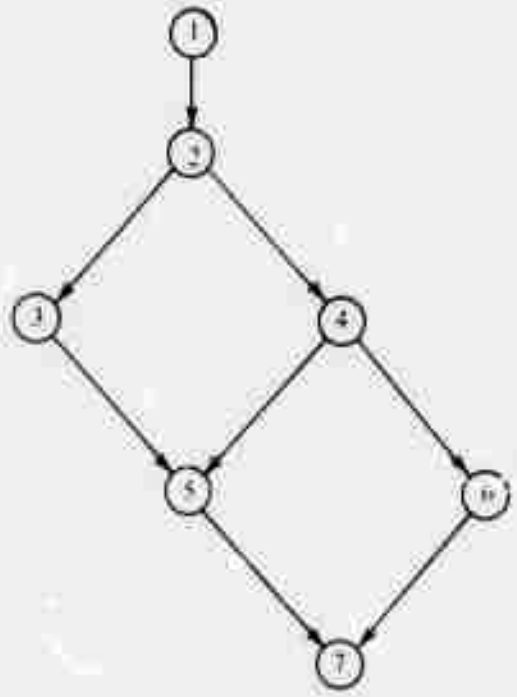


FLOW CHART

PRECEDENCE RELATION

METHOD 2

FIGURE 14. CALCULATION OF X^5+X^3+X



A CONNECTIVITY GRAPH

FIGURE 15

	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1
3	0	0	1	0	0	0	0
4	0	0	0	1	0	1	0
5	0	0	1	0	1	0	0
6	0	0	0	1	0	1	0
7	1	1	1	1	1	1	1

DOMINANCE MATRIX

FOR THE EXAMPLE OF FIGURE 15

FIGURE 16.

operation 3 or operation 4 must be executed. In this case we say that operations 1, 2, and 7 *dominate* operation 5 and that operation 5 is *subordinate* to operations 1, 2, and 7. Operations 5 and 6 are called *exclusive*. This information can be represented by a binary relation, called the dominance relation[30], and by the reachability matrix. Two operations, I and J, cannot both be executed on the same initiation of the process unless either the I, J or the J, I term of the reachability matrix is '1'. If neither term is '1' then I cannot be executed after J and J cannot be executed after I and, therefore, they cannot both be executed for a single initiation of the process. Thus, the reachability matrix specifies whether the execution of two operations is exclusive or not.

To determine, for each operation I, which other operations must be executed when I is executed the dominance relation is constructed. The boolean matrix used to represent the dominance relation is called the *dominance matrix*, D, and has a '1' in the I,J position if and only if I dominates J or the fact that operation J is executed implies that operation I must be executed. Thus, the five possibilities for the execution of two operations, I and J, can be determined as follows where R is the reachability matrix.

D(I, J)	D(J, I)	
'1'	'1'	whenever either operation is executed both of them are executed.
'1'	'0'	whenever J is executed I must be executed but when I is executed J may or may not be executed.
'0'	'1'	whenever I is executed J must be executed but when J is executed I may or may not be executed.
'0'	'0'	If R(I,J) = R(J,I) = '0' then the execution of one implies that the other is not executed. If R(I,J) or R(J,I) is '1' then the execution of one gives no information about the other.

The dominance matrix, D, for a graph with N elements is determined as follows:

1. Set D to all '0's
2. Set I to 0.
3. Set I to I + 1 and find the reachability matrix of the connectivity matrix with row and column I set to '0'.

4. If either R(I,J) or R(J,N) is '0' set D(I,J) to '1' J = 1, 2 . . . N.
5. If I ≠ N go to step 3.

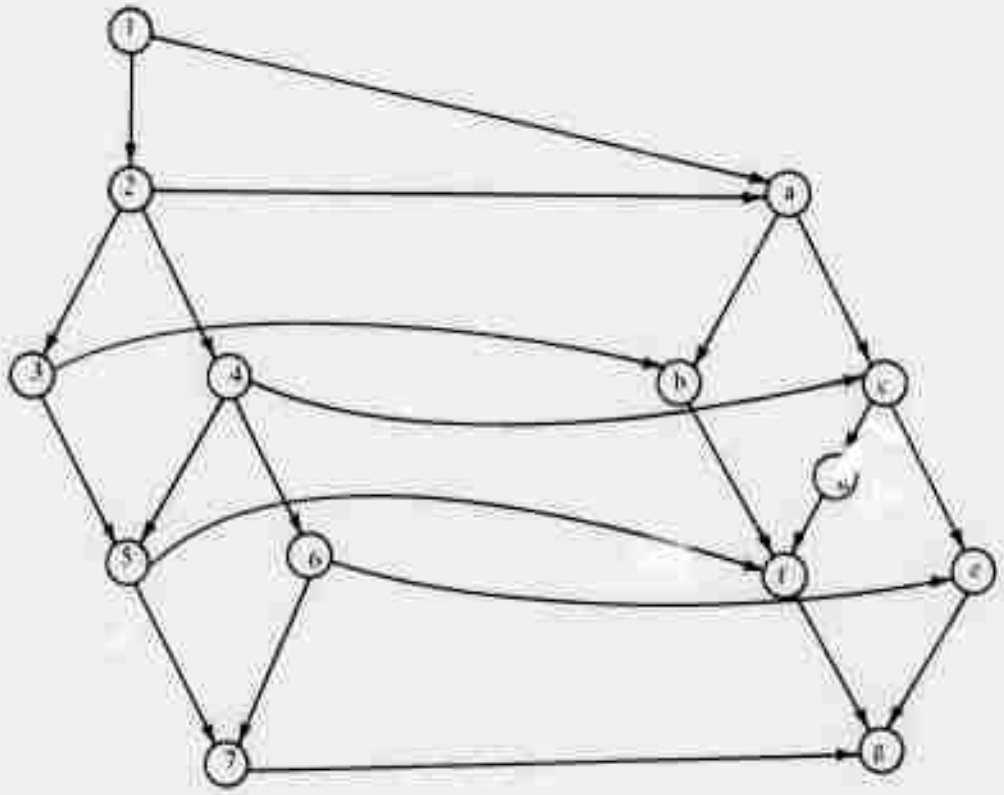
Steps 1 and 2 initialize the D matrix and a counter. In step 3 the reachability matrix, with the Ith operation deleted, is found. If any operation J, cannot be reached from the initiation operation then operation I must be executed every time that operation J is executed since every path from the initiation operation to operation J passes through operation I. Similarly, if any operation, J, cannot reach the termination operation then operation I must be executed every time operation J is. Step 4 sets the appropriate entry of D to a '1' when an operation cannot be reached from the initiation operation or cannot reach the termination operation. Step 5 tests for completion and returns to step 3 if steps 3 and 4 have not been carried out for all values of I. Figure 16 shows the dominance matrix for the example of Figure 15.

2.6 CONTROL GRAPHS

In some cases, where only the nodes on the paths through the connectivity graph are of interest, and not the order of the nodes, a more compact representation called the *control graph* will be used, which represents all paths of the connectivity graph in a more compact form. The control graph is formed from the connectivity graph by combining two nodes if an arc between them is the only extant arc from one of the nodes and the only incident arc to the other node. Also if an arc is extant from a node with more than one extant arc and incident to a node with more than one incident arc it is split into two arcs and a new node inserted between them. Figure 17 shows an example of a connectivity graph and the corresponding control graph. Each node in the connectivity graph can be assigned to a node in the control graph, however, node d of the control graph does not correspond to any node of the connectivity graph. Node d of the control graph represents a path which cannot be specified by any single node in the connectivity graph. If the ratio of decision and merge operations is low the control graph will be much smaller than the connectivity graph. The dominance relation for the operations of a system can be determined from the control graph, usually with less effort than using the connectivity graph, since the control graph will usually have fewer nodes.

In Chapters 3 and 4 we will need to find the directed cut sets of the control graph or of portions of the control graph. The *directed cut sets* of a graph are defined as all sets of nodes such that no member of a set can reach any other member of that set and such that removal of all nodes of a set will leave no path from the initiation node to the termination node. Figure 18 shows the directed cut sets for the example of Figure 17. The nodes in each directed cut set represent a set of exclusive possible paths for the control each time the process is initiated. The operations corresponding to exactly one of the nodes in each directed cut set will be executed when the process is executed. The algorithm shown in Figure 19 gives a method of determining the directed cut sets of an N node graph from its control matrix. V is an array of boolean vectors, each vector having a position for each node of the control graph. R(I) is an N element boolean vector with a '1' in position I if node I can reach or be reached by node I. "NODE" is an array of integers used to record node numbers.

The algorithm operates by systematically selecting one node of the graph and deleting all nodes that can reach or be reached by that node. This process is repeated until there are no nodes remaining, in which case all of the nodes



CONNECTIVITY GRAPH

CONTROL GRAPH

FIGURE 17.

A CONNECTIVITY GRAPH AND THE CORRESPONDING CONTROL GRAPH

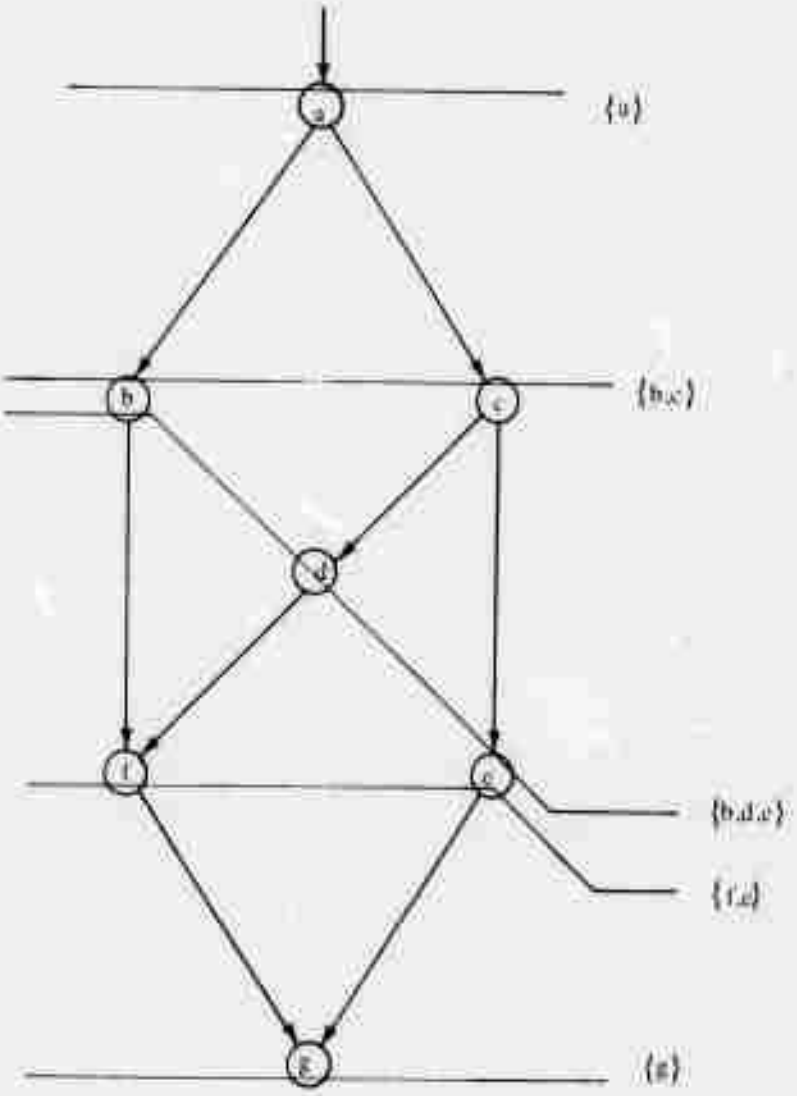


FIGURE 18. DIRECTED CUT SETS FOR THE CONTROL GRAPH OF FIGURE 17

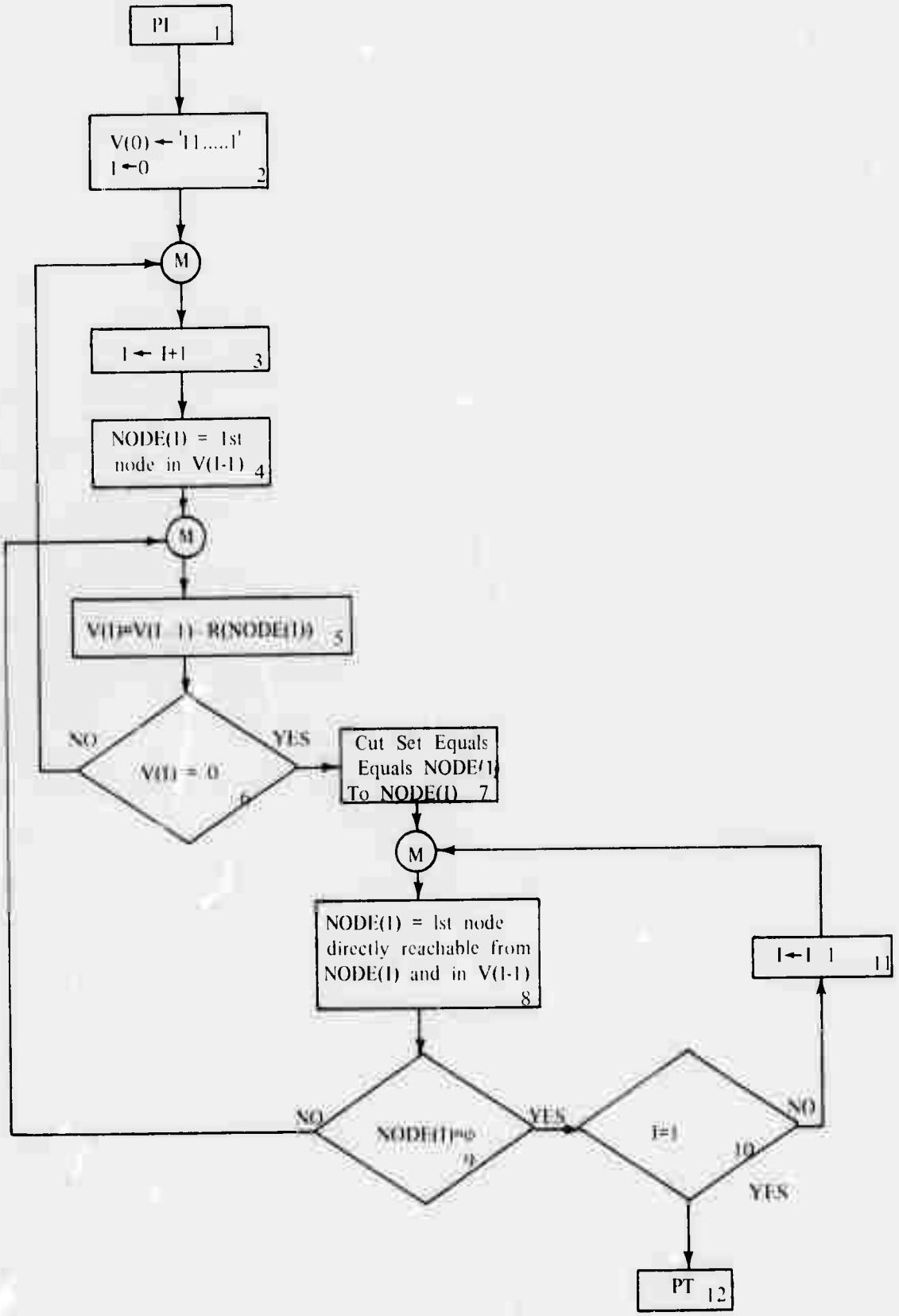


FIGURE 19.

ALGORITHM FOR CALCULATING THE DIRECTED CUT SETS

that have been selected to that point form a directed cut set. Step 2 initializes V(0) to all '1's and the V array is used to record which nodes have been deleted. I records the number of nodes that have been deleted. In step 4 the first node that was not deleted at the preceding level is found and in step 5 the nodes reaching and reachable from this node are deleted. Step 6 tests to determine if all nodes have been deleted and if they have not, control returns to step 3 to find and delete another node. If the test at step 6 finds that all nodes have been deleted, step 7 records the selected nodes as a cut set. Step 8 finds a node in the last set of nodes deleted that can be used instead of the selected node, and that directly follows it. If one exists, control returns to step 5 to delete the nodes reaching and reached by it. If no such node exists then all cut sets that include NODE (I, J) have been found. If I is not '1' then I is decremented and the algorithm goes to step 8 to test for a new node at the preceding level. If I is '1' then all cut sets have been found. Figure 20 illustrates graphically the process by which the nodes are selected and the cut sets determined.

2.7 SUMMARY

Several binary relations and methods for calculating them have been discussed in this chapter. The precedence relation specifies any required execution sequence, the dominance relation specifies which operations must be executed if a given operation is executed, and the control graph is used to determine sets of operations which form directed cut sets for a process. These relations will be used in the following chapters to aid in the synthesis of concurrent processes.

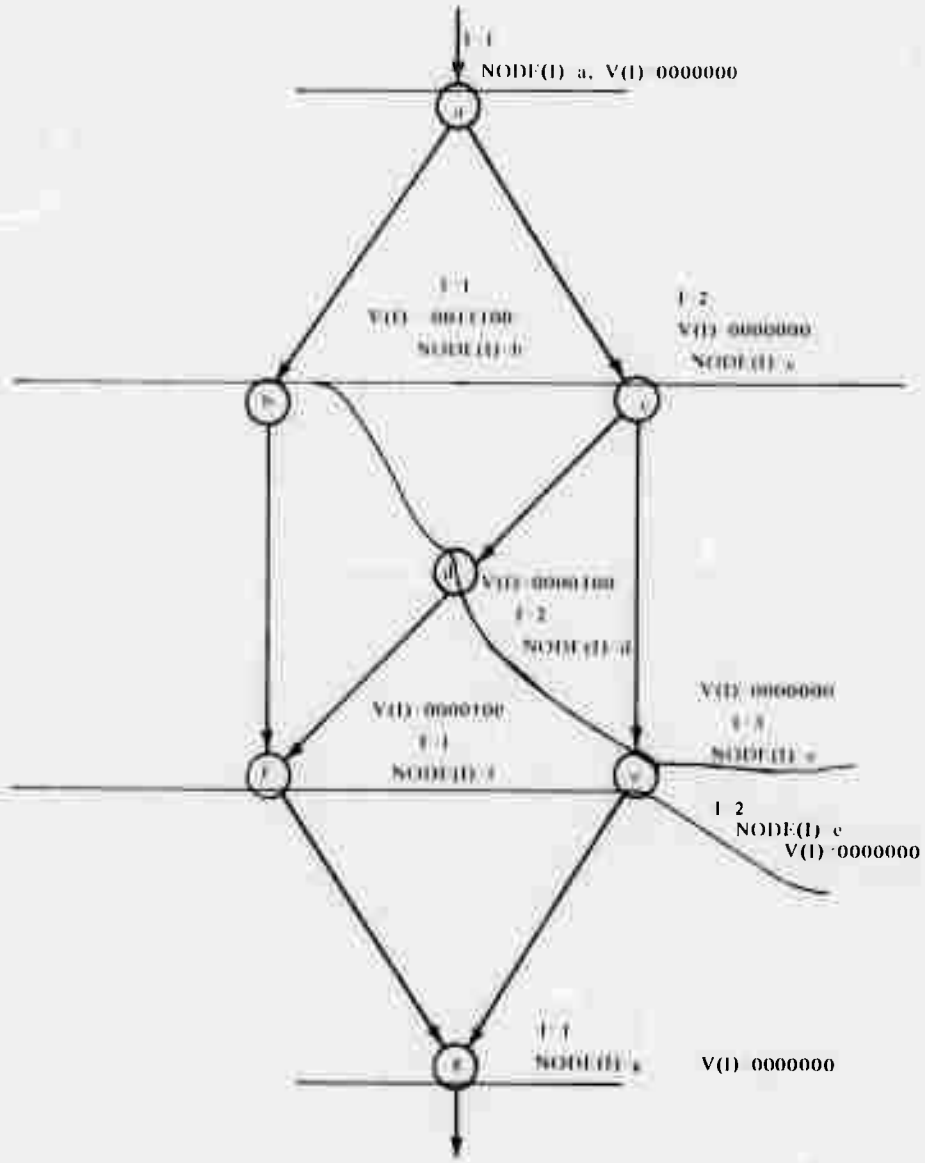


FIGURE 20.
AN EXAMPLE OF THE
DETERMINATION OF CUT SETS

3. LOOP-FREE PROCESS SYNTHESIS

Once the relations discussed in Chapter 2 have been determined, the next step is to use this information to synthesize an *error-free* control network in which each operation is initiated as soon as all of its predecessors are completed. The types of errors considered are discussed in Section 3.1 and Sections 3.2 and 3.3 consider the synthesis of error-free processes. Chapter 4 extends this work to include processes which contain loops.

3.1 TYPES OF ERRORS CONSIDERED

The two main categories of errors considered are sequencing errors which may be prevented by satisfying the precedence relation, and implementation errors. Implementation errors are caused by incorrect use of branch and rendezvous operations. The types of implementation errors considered are *hazards*, where an operation may be initiated before it has generated the completion signal from a preceding initiation, and *incomplete rendezvous*, where at least one but not all inputs to a rendezvous receive initiation signals. The hazard and re-entered branch errors described by Keller[23] have been combined into the single category of hazard since the re-entered branch error is a special case of a hazard. Figures 21 and 22 show examples of processes with errors. The process shown in Figure 22 will operate correctly the first time it is initiated but one of the rendezvous operations will receive an initiation signal on only one input and will be waiting for an initiation signal on the other input. If the other decision outcome is taken when the network is initiated the second time, the rendezvous will have received a previous initiation signal on the input reached by operation 2 and will produce its completion signal when it receives the initiation signal from the decision. This would allow the operation following the rendezvous to be initiated before one of its predecessors. To insure error-free performance of the process in this example, the output from operation 2 must be rendezvoused with the outputs from decision operation 3 before operation 4 or 5 is initiated. However, the outputs from decision operation 3 must be merged together before they can be rendezvoused with the termination signal from operation 2. Figure 23 shows a process with no errors that achieves the desired operations in which a new register, G, has been employed. The added operations to set and test G are required so that the outputs from decision 3 can be merged together and then split, after rendezvousing with the output from operation 2.

To simplify the process shown in Figure 23, a new operation, called the *wait operation*, will be introduced. It has one input defined as the signal input and two or more inputs defined as decision inputs, with an output corresponding to each of the decision inputs. Whenever an initiation signal is applied to the signal input and one of the decision inputs, a completion signal is generated on the corresponding output. A wait operation functions as a group of two-input rendezvous operations which share one input signal and which are all cleared whenever an output signal is generated by any one of them. Thus, a wait operation can be used to replace all of the operations shown enclosed in the dotted box in Figure 23.

3.2 DECISION FREE PROCESSES

Before discussing the general technique for determining a control network, consider the example in Figure 24. There are eight processing operations with an assumed precedence relation as shown and no decision operations.

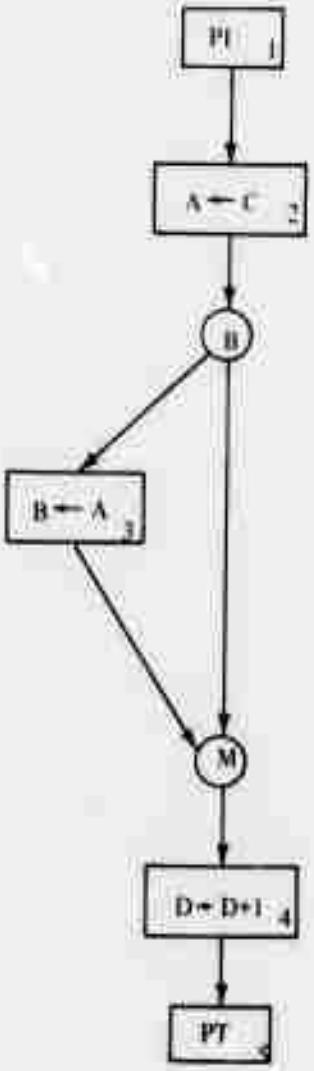


FIGURE 21.
A PROCESS WITH A HAZARD

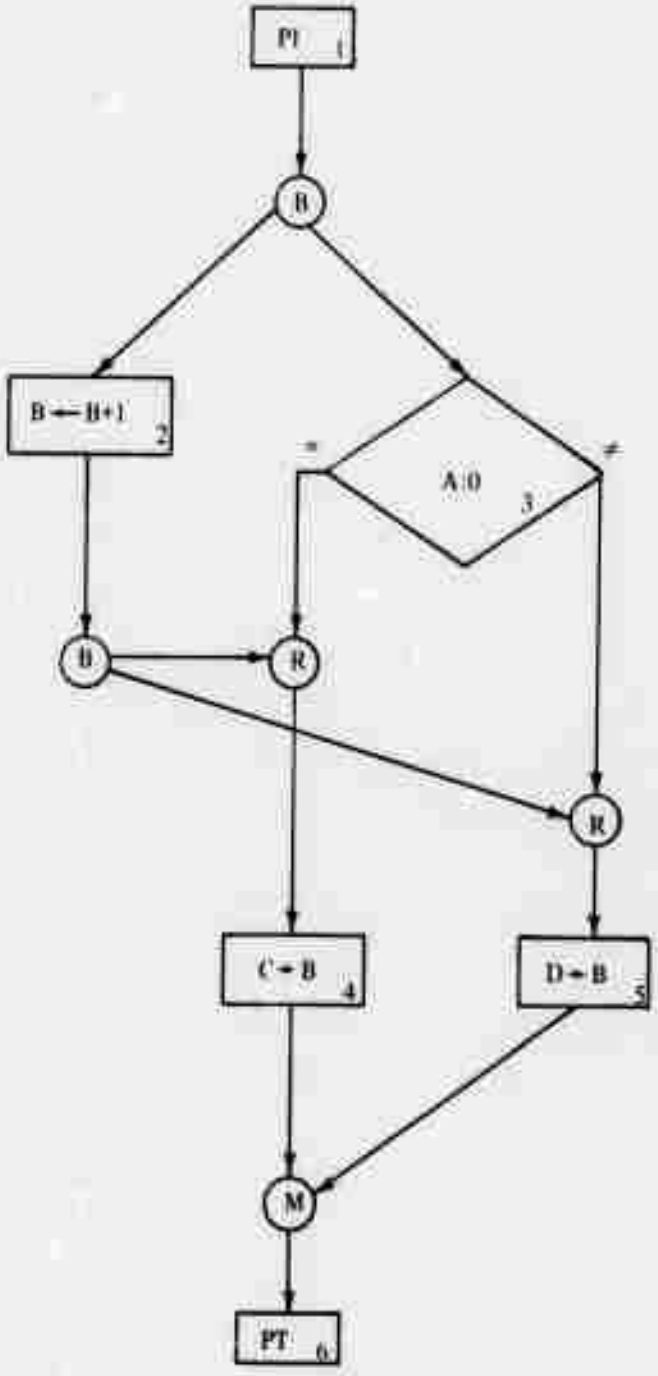


FIGURE 22.

A PROCESS WITH AN INCOMPLETE
RENDEZVOUS

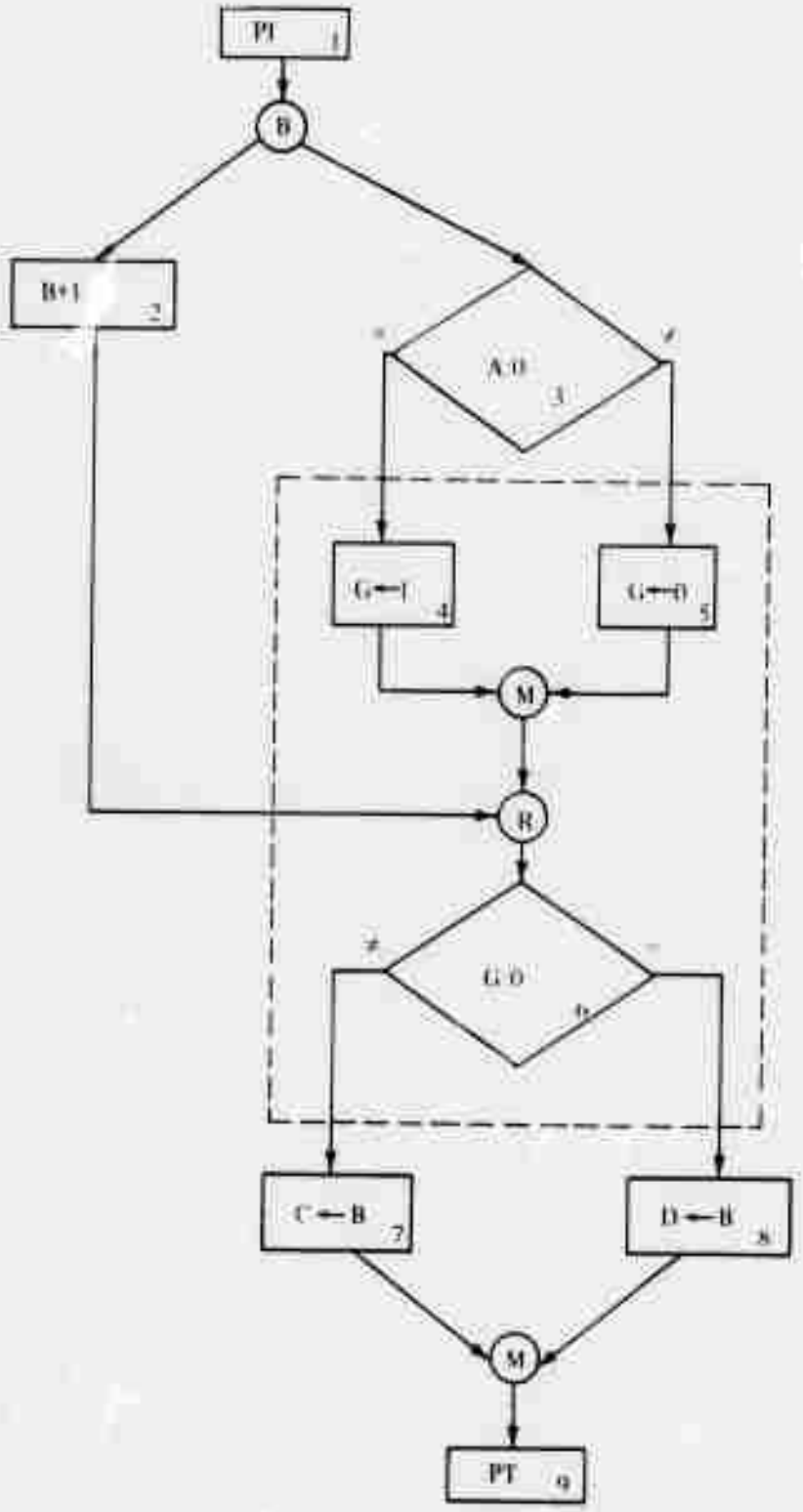
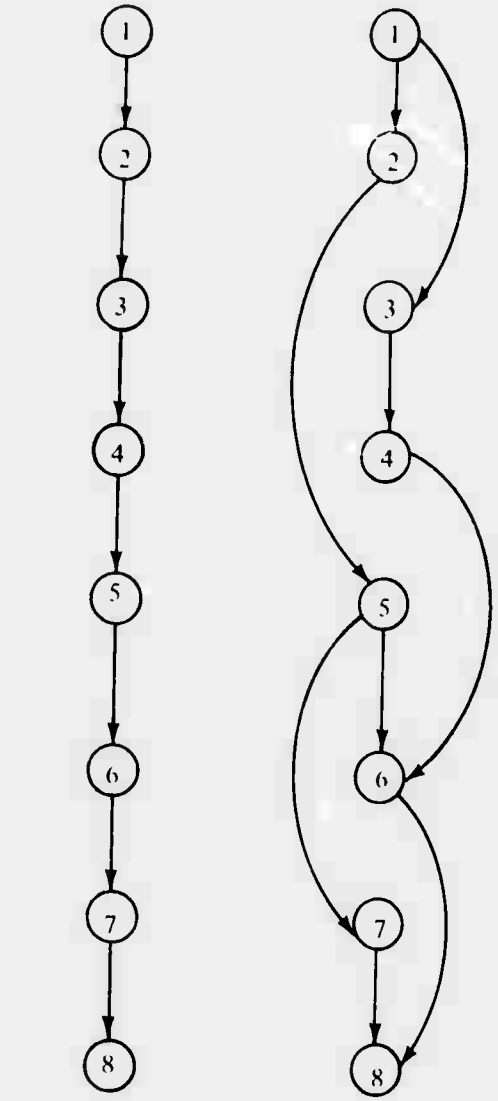


FIGURE 23.

THE PROCESS OF FIGURE 22 WITHOUT THE INCOMPLETE RENDEZVOUS



CONNECTIVITY GRAPH

PRECEDENCE GRAPH

FIGURE 24. A DECISION-FREE PROCESS

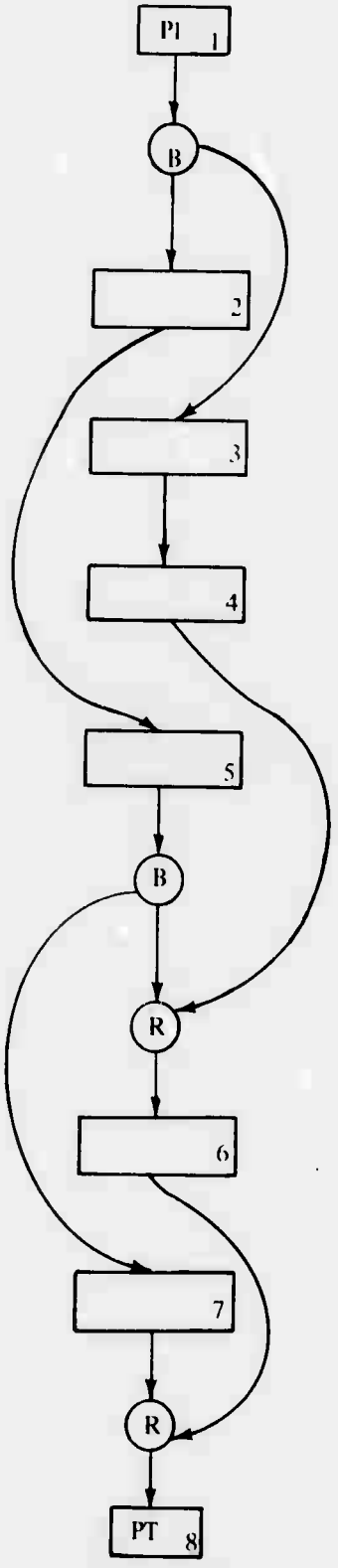


FIGURE 25. CONCURRENT FLOW CHART FOR THE EXAMPLE OF FIGURE 24

Each operation can be initiated when all of its direct predecessors are completed and the flowchart for an equivalent concurrent process is shown in Figure 25. The following theorem describes a method which can be used to synthesize the concurrent process and a proof that the synthesized process will be error free.

Theorem 1: The following three steps specify the synthesis of an error-free process, in which every operation is initiated as soon as all of its predecessors are completed, given the precedence relation of a decision-free sequential process.

a. Connect a branch operation to the output of each operation, J, with one branch operation output for every operation of which operation J is a direct predecessor. If J is the direct predecessor of only one operation then no branch operation is required for J.
b. Connect a rendezvous operation to the input of each operation, I, with an input for each operation which is a direct predecessor of I. If I has only one direct predecessor then no rendezvous operation is required for J.
c. For each operation, I, connect one output of the branch operation associated with each predecessor of I or the predecessor output if it has no associated branch operation, to an input of the rendezvous for I or to I's input if I has no associated rendezvous operation.

Proof: Let the operations of the process be numbered sequentially in the order of their execution in the sequential process, with the PI operation numbered 1. Then all predecessors of operation K must be assigned numbers that are less than K since operation K cannot be required to follow an operation that it does not follow in the sequential process. For any operation, K, assume that sometime after the initiation of the PI operation each direct predecessor of K generates exactly one completion signal. Then each input to the rendezvous operation preceding operation K will receive exactly one initiation signal and therefore, the rendezvous will generate one completion signal as soon as all the direct predecessors of K are completed. The rendezvous completion signal will be used to initiate operation K and so operation K will receive exactly one initiation signal. Thus, if all direct predecessors of operation K are executed exactly once there are no hazards or incomplete rendezvous in operation K and the rendezvous associated with it. Since the PI operation is initiated exactly once and is the only direct predecessor of operation 2, operation 2 must be executed exactly once with no hazard or incomplete rendezvous errors in it or its associated rendezvous operation. Then by induction, operation K for K greater than or equal to one must be initiated exactly once and have no hazard or incomplete rendezvous errors. Thus, all operations are free of implementation errors and will be initiated as soon as their predecessors are completed.

Thus, for a sequential process without decisions, a concurrent process can be developed very easily. In fact it is quite interesting to observe that if each rendezvous operation is combined with the operation following it and each branch operation is combined with the operation it follows, the resulting flow chart is isomorphic to the basic precedence graph.

3.3 PROCESSES CONTAINING DECISIONS

Figure 26 shows a more complicated process which contains a decision. Here the outputs from the direct predecessors of each operation cannot be rendezvoused since this will result in implementation errors. In this example, the direct predecessors of operation 5 are operation 3 and one of the decision outputs. These cannot be rendezvoused, however, because the output from operation 3 will occur every time the process is initiated while the decision output will not occur every time. In this case operation 3 is executed for some initiations of the process for which its successor is not initiated, or operation 3 is not dominated by its successor. Operation 7 represents a different case since it is not dominated by its predecessor. There are four possible dominance relations between an operation, I, and one of its predecessors, J, and these are listed below where $I > J$ means that I dominates J, and $I \not> J$ means that I does not dominate J.

1. $I > J,\ J > I$
2. $I > J,\ J \not> I$
3. $I \not> J,\ J > I$
4. $I \not> J,\ J \not> I$

In case one, operation I and its predecessor are both executed under the same conditions so that the output from the predecessor, J, can be rendezvoused with other signals and used to initiate operation I. In case two, the predecessor is not executed every time operation I is executed so its completion signal must be merged with some other signal before it can be used to initiate operation I. In case three, operation I is not executed every time its predecessors is, so the completion signal from J must be separated into two exclusive parts, one that is subordinate to I and one that occurs only when I is not to be executed. Case four is a combination of cases two and three. The output of the predecessor, J, must first be split into exclusive parts and then the part that is subordinate to I must be merged with other signals before being used to initiate I. In case four it would be possible to first merge the completion signal from J with other signals and then split the merged output, but the former approach is simpler.

3.3.1 Direct Predecessors With Conditions Equal To Their Successor

In this case each direct predecessor is executed if, and only if, the operation is to be executed. This is the same as the decision-free case and the completion signals from the direct predecessors can be rendezvoused together to generate the initiation signal for the operation. In fact, in any case where two or more direct predecessors have the same conditions (they dominate each other) their completion signals can be rendezvoused together and the rendezvous output considered as the output of a single direct predecessor with the same dominance relation as the operations.

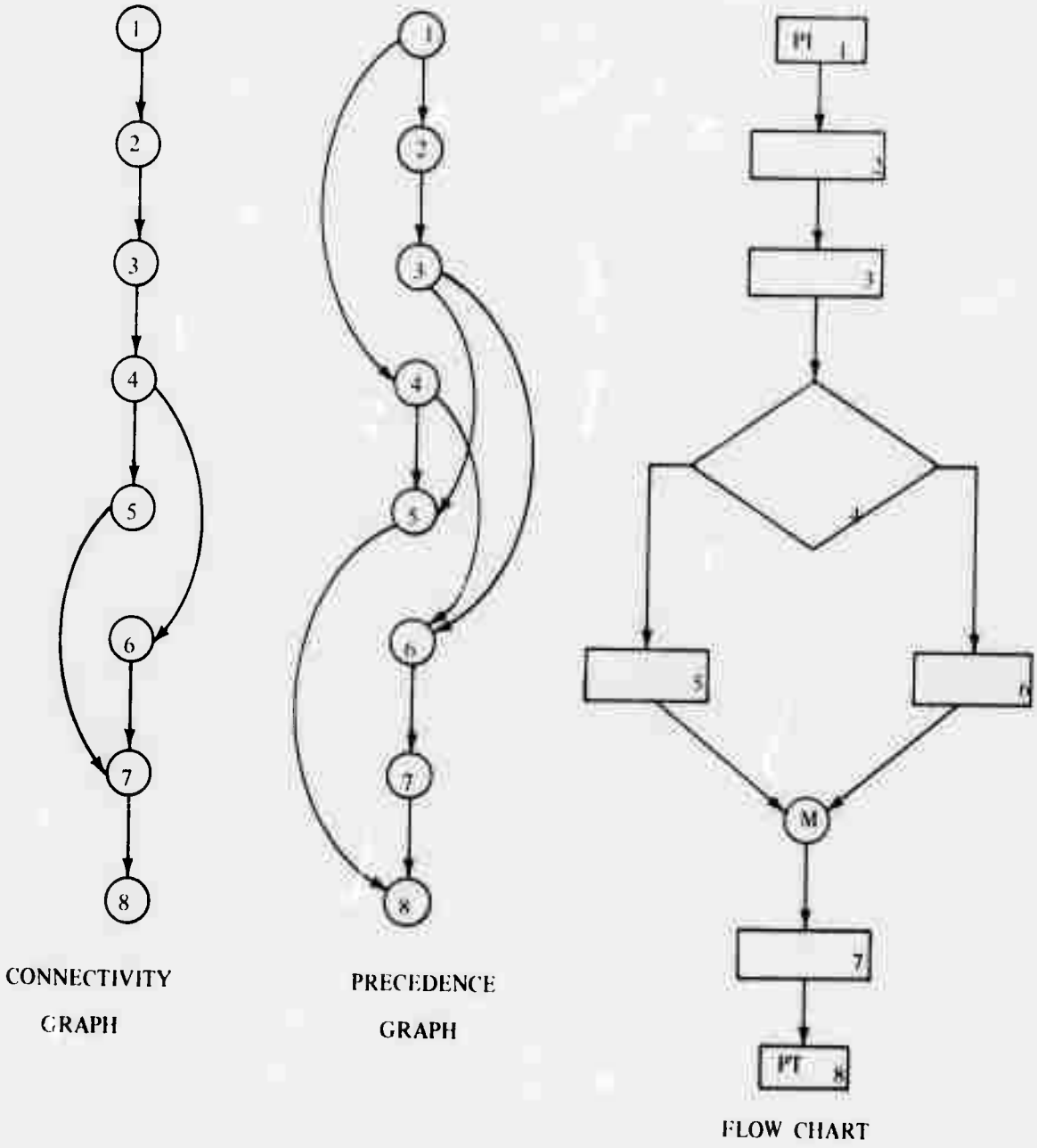


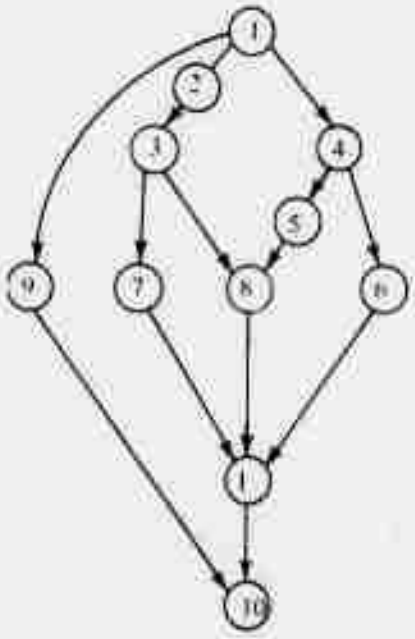
FIGURE 26. EXAMPLE OF A PROCESS WITH A DECISION

3.3.2 Direct Predecessors That Do Not Dominate Their Successor

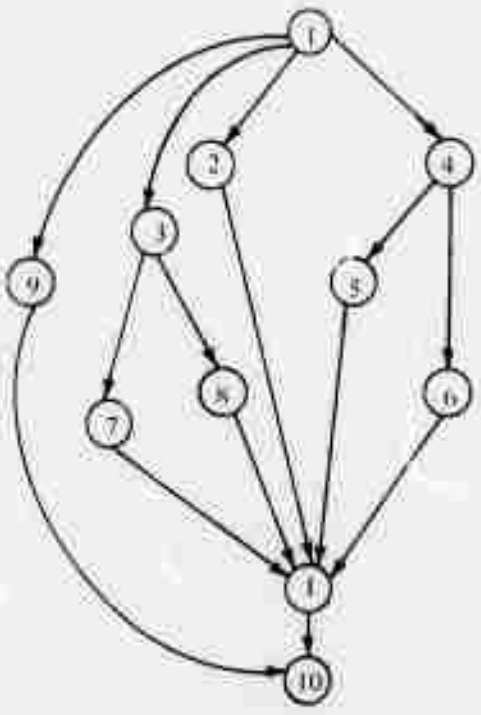
Next consider the direct predecessors of an operation, I, that are subordinate to I but do not dominate I. These predecessors are never executed unless I is to be executed, but I is executed when some of the predecessors are not executed. Therefore, the completion signals cannot be rendezvoused together as in Section 3.3.1 since some of them will not be executed every time the others are executed. An example of this situation is shown in Figure 27 where operation I has operations 2, 5, 6, 7, and 8 as direct predecessors. Since exactly one of the operations 2, 5, or 6 will be executed when the system is initiated, the completion signals from these operations can be merged together. This merged signal will be generated whenever operation I is to be executed and the one direct predecessor 2, 5, or 6 that is executed, has been completed. Similarly the completion signals from 6, 7, and 8 can be merged together to produce a signal when one of them is completed. These two signals can then be rendezvoused together to generate the initiation signal for operation I as shown in Figure 28. Each of the inputs to the rendezvous is generated if and only if I is to be initiated [illegible] are generated only when all predecessors of I are completed. Thus, I is initiated as soon as all of its direct predecessors are completed.

A general method is now presented that can be used to determine the combinations of signals that can be merged together to form the inputs to the rendezvous operation. Each merge operation must have inputs from a set of operations, only one of which can be executed for a single initiation of the system, and one of which will be executed if the operation is to be executed. The combinations of conditions that can be merged together are determined by finding the directed cut sets of the portion of the control graph that is dominated by and reaches the node corresponding to operation I. A *directed cut set* of the control graph is a set of nodes whose deletion removes all paths from the initiation node to the termination node and such that none of the nodes is reachable from any other node in the cut set, as discussed in Section 2.6. The directed cut sets for the control graph of the example of Figure 27 are shown in Figure 29. The operations represented by different nodes in a cut set are just the operations whose outputs can be merged together to form an input signal for the rendezvous operation, since the operations represented by exactly one of the nodes in the cut set will be executed each time I is to be executed. Since there are numerous cut sets the completion signals of the direct predecessors can be merged in a number of different combinations.

A cut set table which is similar to a prime implicant table[47] can be used to select the combinations of merge unit inputs to be used. The table will have a row for each possible merge unit (cut set of the control graph) and a column for each direct predecessor. The standard techniques which have been developed for prime implicant tables[47, 48] can be applied to the cut set table, such as identifying essential terms, to select a set of rows that cover all of the columns. One merge is then required for each row of the cut set table that is selected. Figure 30 shows the cut set table to be used with the cut sets shown in Figure 29. The fifth row is essential to cover the last column and row 3 covers the remaining columns. The M-R network corresponding to this covering of the cut set table is shown in Figure 28. There is no direct predecessor of operation I corresponding to node C of the control graph but the required completion signal could be obtained from the decision output that reaches node C if required. Thus an M-R network can always be synthesized to generate the initiation signal for an operation when the operation dominates its direct predecessors.



CONNECTIVITY GRAPH



PRECEDENCE GRAPH

FIGURE 27. EXAMPLE OF AN OPERATION THAT IS NOT DOMINATED BY ITS PREDECESSORS

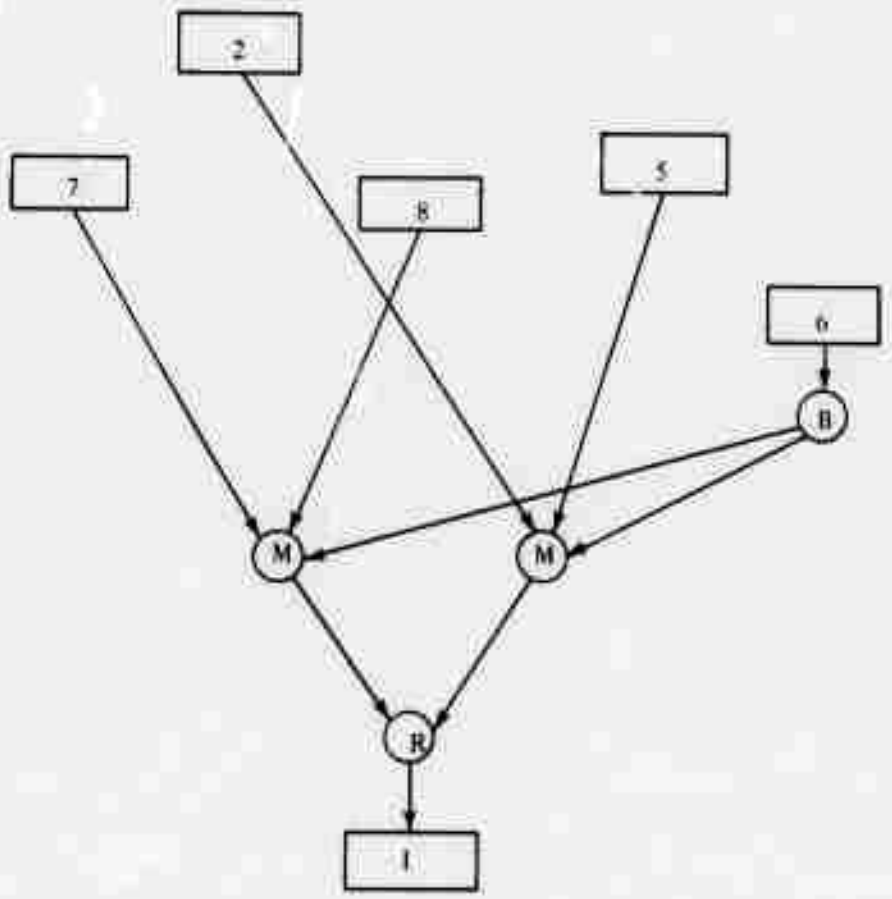
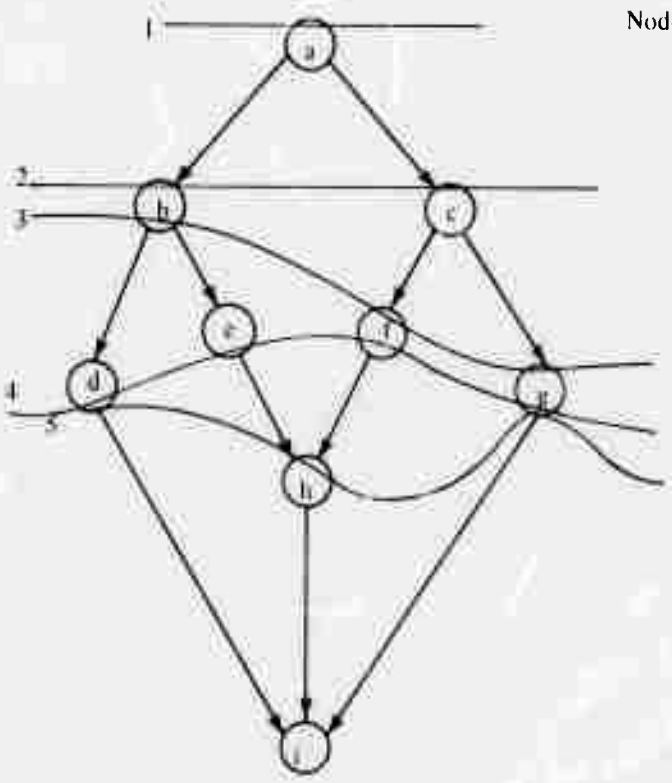


FIGURE 28. FORMATION OF THE INITIATION SIGNAL FOR OPERATION 1 OF FIGURE 27



Node of Control Graph	Corresponding Nodes in Connectivity Graph
a	1
b	2,3
c	4
d	7
e	
f	5
g	6
h	8
i	1

FIGURE 29. THE PORTION OF THE CONTROL GRAPH REACHING i AND THE CUT SETS FOR THE EXAMPLE OF FIGURE 27

CUT SET	DIRECT PREDECESSOR 2	5	6	7	8
1					
2	X				
3	X	X	X		
4		X	X	X	
5			X	X	(X)

CUT SET TABLE FOR THE EXAMPLE OF FIGURE 27

FIGURE 30

The M-R networks that are synthesized are similar to two-level or-and switching networks. In some cases more economical networks may be synthesized by using more than two levels as described in Miller[49], or by considering the requirements for several operations simultaneously and using techniques that have been developed for the synthesis of multiple output switching networks[47, 49].

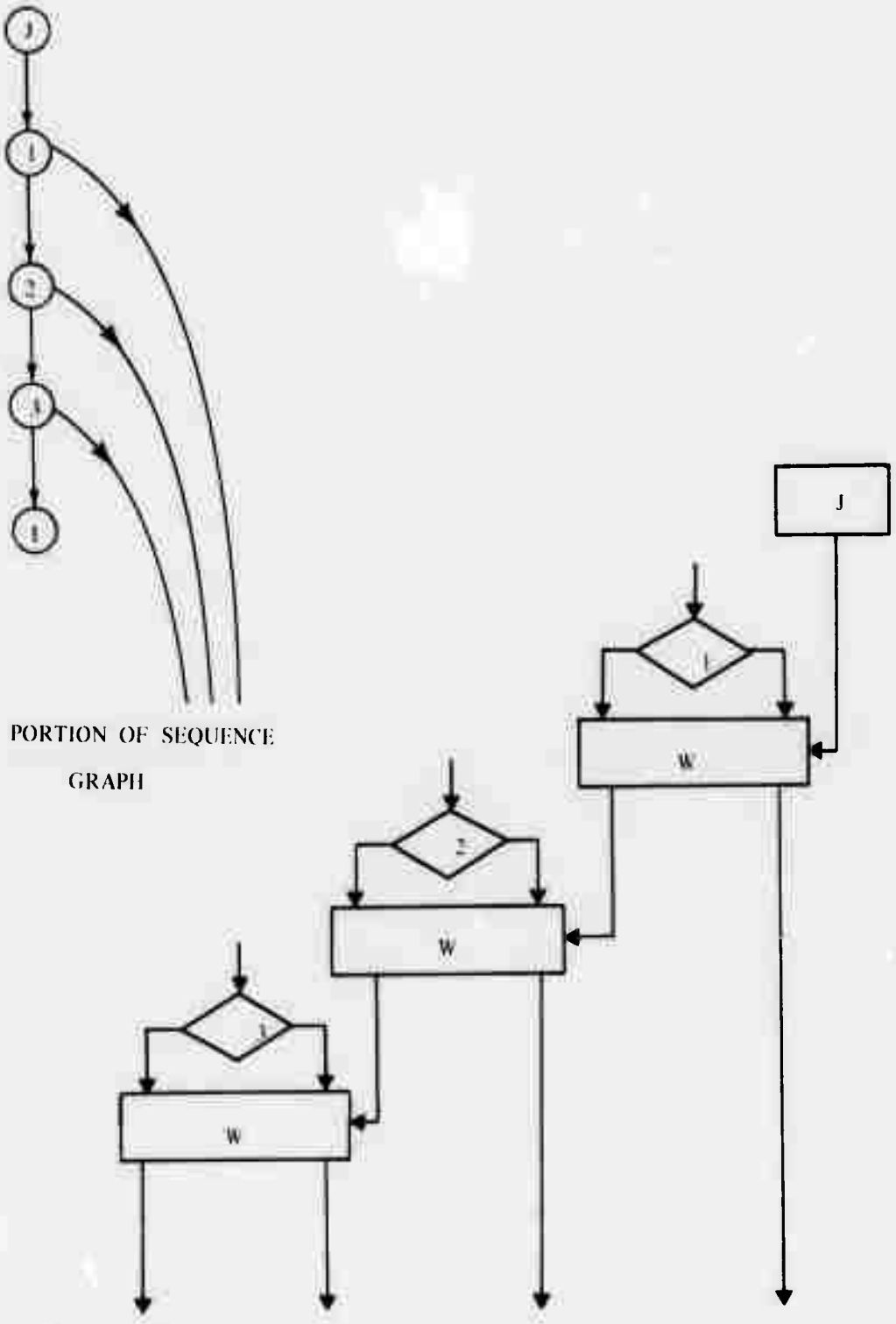
3.3.3 Direct Predecessors That Are Not Subordinate To Their Successor

Finally, consider the case where, J, a predecessor of I, is not subordinate to I. This means that J will be executed for some cases where I is not. Since J is a predecessor of I there must be a path from J to I in the sequence graph and since I does not dominate J there must be some path, from J to the termination node, that does not reach I. Then there must be at least one decision in the sequence graph that is reached by J, that has an output that reaches and is subordinate to I, and that has an output that does not reach I. Find one such decision, D, that does not reach any other decision having these characteristics. D must be a predecessor of I. Add a wait operation with a decision input from each output of D. Then consider the signal input to the wait operation to be the input to an operation that has J as a direct predecessor and that has the same dominance relation as decision D. The output signal(s) from the wait operation that correspond to the decision output(s) that is (are) subordinate to I, will be generated only after J and D are completed and only if I is to be executed, therefore, they can be used as inputs to the M-R network for I. They can be treated as completion signals from operations that are direct predecessors of operation I and that have dominance relations equal to those of the corresponding decision outputs. Thus, completion signals from direct predecessors that are not subordinate to their successor can be split in this manner into signals that are exclusive of and subordinate to their successor. Then the methods described in Section 3.3.2 can be used to synthesize the M-R network to initiate the operation.

The signal inputs to any wait operations that have been added must be synthesized. They will be subject to the same possibilities as the inputs to the operations for which the wait operations were added. However, there will be one less decision between the wait operation input and its direct predecessor than there was between the original operation and the direct predecessor. The process of adding wait operations may be repeated a number of times for one direct predecessor of the original operation but only a finite number need be added since there can only be a finite number of decisions between operations I and J. Figure 31 shows an example where three wait operations must be used to split the completion signal from operation J into parts that are exclusive of and subordinate to operation I. In this case the three wait operations could be combined into a single four-output wait operation as shown in Figure 32. This possibility will not be considered further, however, since the wait operation outputs that are eliminated by this network may be required by other operations and since testing for its occurrence would complicate the method for splitting the completion signals from the predecessors. A scan of the complete system, after the initiation networks for all operations have been synthesized, might be used to check for cases where wait operations can be combined.

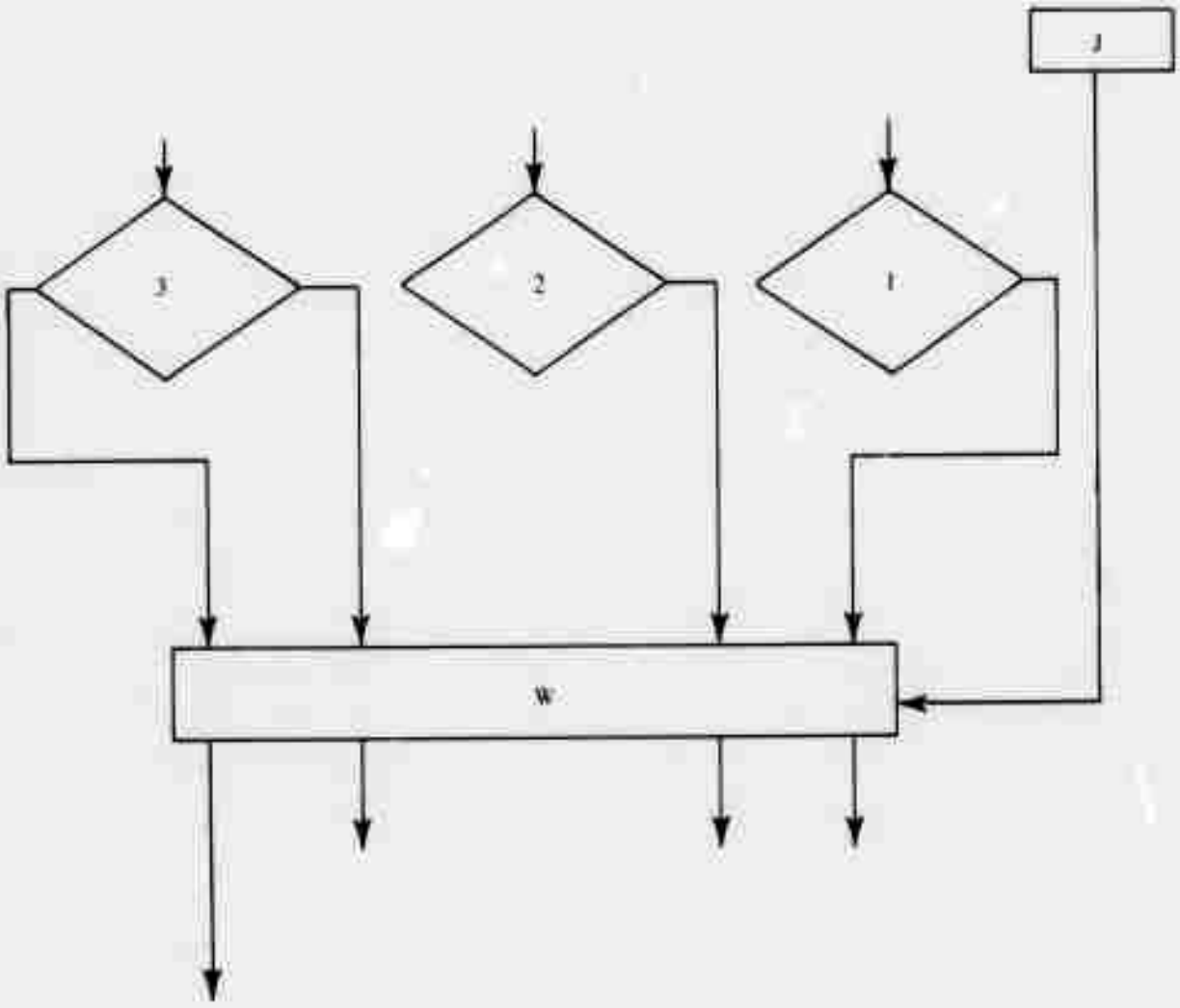
3.3.4 Conditional Direct Predecessors

One problem that arises when using the methods discussed in the preceding sections is illustrated in Figure 33. Operation 5 depends on operation 2 but this is not shown explicitly in the basic precedence graph since it is implied



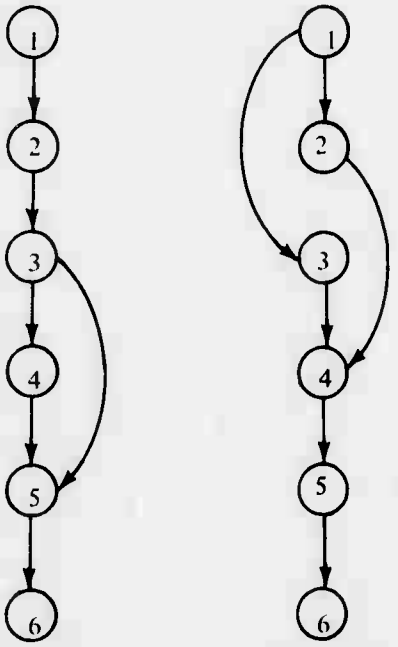
WAIT OPERATIONS REQUIRED TO SEPARATE COMPLETION SIGNAL FROM OPERATION J

FIGURE 31. EXAMPLE OF THE USE OF THE WAIT OPERATION



TO OPERATION 1

FIGURE 32. COMBINATION OF THE 3 WAIT OPERATIONS OF FIGURE 31 INTO A SINGLE WAIT OPERATION



CONNECTIVITY GRAPH

PRECEDENCE GRAPH

FIGURE 33. A PROCESS WITH A CONDITIONAL DIRECT PREDECESSOR

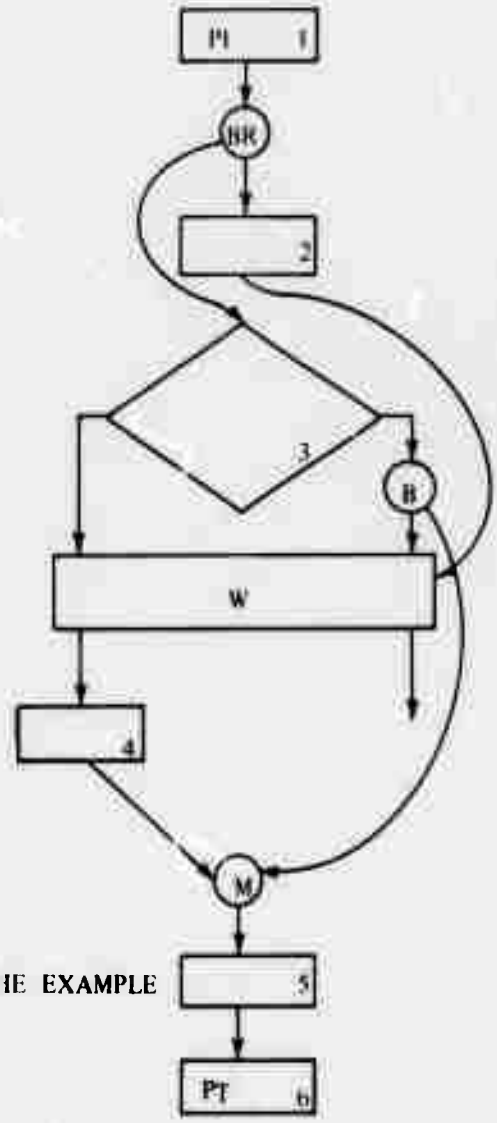


FIGURE 34. CONCURRENT PROCESS FOR THE EXAMPLE OF FIGURE 33

by the dependence of operation 5 on operation 4 and of operation 4 on operation 2. Figure 34 shows the control network which would be synthesized using the above techniques. This network has several faults. First, the wait operation has an unused output which could still be active after initiation of the PT operation. Second, if the decision bypasses operation 4, then operation 5 can be initiated before operation 2 is completed, and the completion signal for the entire network may be generated before operation 2 is completed. These problems arise because the execution of operation 4 depends on the outcome of decision 3.

In the previous discussions we have assumed that an operation, I, may be initiated when all of its direct predecessors are completed. Since the direct predecessors of I require the execution of all other predecessors of I to be completed before they are initiated, the execution of all predecessors of I should be completed when the completion signals from the direct predecessors of I are generated. The problem which occurs in the example of Figure 33 is that for one outcome of decision 3, operation 4, which is the direct predecessor of operation 5, is not executed. In this case operation 2 is a *conditional direct predecessor* of operation 5. That is, operation 2 is an indirect predecessor of operation 5 but in one of the decision outcomes operation 2 is not a predecessor of any *executed* predecessor of 5. Operation 2 must be considered a direct predecessor of operation 5 for the decision outcome which bypasses operation 4. In general a conditional direct predecessor of an operation, I, is an indirect predecessor which for some conditions or combinations of decision outcomes is not a predecessor of an *executed* predecessor of operation I. Conditional direct predecessors can be eliminated by modifying and making additions to the precedence relation to take account of the wait operation. In the example of Figures 33 and 34 the wait operation is added when the initiation signal for operation 4 is synthesized. This wait operation can be considered as two additional operations in the process, one for each of the wait operation outcomes. These operations are labeled $W_{1,1}$ and $W_{1,2}$ and added to the direct precedence graph as shown in Figure 35. They have as predecessors decision 3 and operation 2 which supply the input signals for the wait operation. Their successors are operation 4, for which the wait operation is added, and the successors of operation 4. The direct successor of $W_{1,1}$ is operation 4 but operation 5 is the direct successor of $W_{1,2}$ since $W_{1,2}$ does not reach operation 4. Thus, to its predecessors the wait operation is treated as a single operation while to its successors it is treated as a separate operation for each of its outputs. With this interpretation of the wait operation, the direct predecessors of each operation are all dominated by the operation after all required wait operations have been added. Using the precedence graph of Figure 35 the error-free concurrent process shown in Figure 36 is synthesized.

The following rules are used to modify the precedence relation when a wait operation, W_i, is added because a direct predecessor, J, of operation I is not dominated by I.

1. Operation J, the decision supplying the decision inputs, and all of their predecessors are made predecessors of W_i. Also, if any predecessors of W_i are wait operation outcomes then any other outcomes from the same wait operation that reach W_i are made predecessors of W_i also.
2. I and its successors are made successors of each outcome of W_i that can reach them.

These changes can easily be made to the complete precedence relation but only those additional members of the precedence relation that are not implied by other members can be added to the basic precedence relation. In fact, it may be necessary to delete some of the original members of the basic precedence relation when the members due to

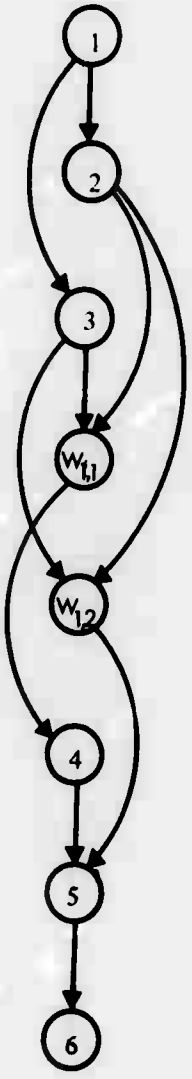


FIGURE 35. PRECEDENCE GRAPH FOR THE EXAMPLE OF FIGURE 34 INCLUDING THE WAIT OPERATION

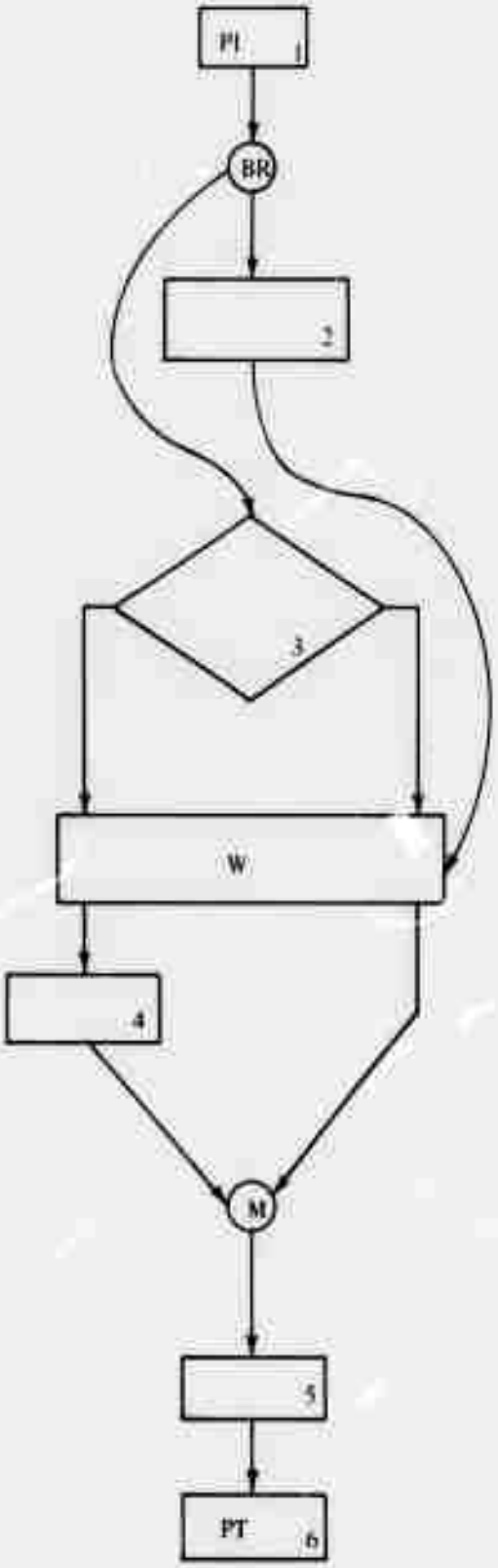


FIGURE 36. ERROR-FREE CONCURRENT PROCESS FOR THE EXAMPLE OF FIGURE 33

a new wait operation are added. In order to determine the basic precedence relation with an added wait operation we can add all members of the complete precedence relation that include the wait operation to the basic precedence relation, calling this the augmented precedence relation, and then delete any implied precedence relations.

Determination and deletion of the implied relations when a single operation is added or when several operations which cannot be predecessors of each other are added is much simpler than the general case described in Section 2.2.2 since no member of the relation can be implied by more than two other members. Thus all implied relations that are in the augmented precedence matrix can be found by merely multiplying the augmentated matrix by itself. The result of multiplying an augmented precedence matrix by itself is shown below where P represents the original precedence matrix.

$$\begin{bmatrix} P & B \\ C & D \end{bmatrix} \quad \begin{bmatrix} P & B \\ C & D \end{bmatrix} \quad \begin{bmatrix} P \cdot P + B \cdot C & P \cdot B + B \cdot D \\ C \cdot P + D \cdot C & P \cdot B + D \cdot D \end{bmatrix}$$

The portion of the augmented precedence matrix labeled D must be zero since there can be no member of the precedence relation between two outcomes of the same wait operation. Therefore all of the terms involving D can be dropped to give the following matrix.

$$\begin{bmatrix} P \cdot P + B \cdot C & P \cdot B \\ C \cdot P & C \cdot B \end{bmatrix}$$

Also since there are no members of P that are implied by two other members of P it is not necessary to calculate P P and since D is zero it is not necessary to calculate C B. Thus the following matrix specifies all of the members of the augmented precedence relation which must be deleted in order to make it a basic precedence matrix.

$$\begin{bmatrix} B \cdot C & P \cdot B \\ C \cdot P & 0 \end{bmatrix}$$

The following theorem shows that all predecessors of each operation will be completed before the operation is initiated.

Theorem 2: Adding all wait operation outcomes to the precedence relation insures that no operation is executed until all of its predecessors are completed.

Proof: Let I be executed for a particular initiation of the process and let J be any predecessor of I that is also executed. If J is a direct predecessor of I then J must be completed before operation I is initiated. If J is not a direct predecessor of I it must be the predecessor of

some direct predecessor, K, of I. We will show that there is a chain of operations I, K_n, K_{n-1}, . . . , K_1, J where K_n is an executed direct predecessor of I, J is a direct predecessor of K_1, and K_{i-1} is an executed direct predecessor of K_i for i = 2, 3, . . . n.

Let Q be any operation that is a predecessor of I and that has J as a direct predecessor. There must be one such operation unless J is a direct predecessor of I. Q is either a wait operation outcome or not. If it is not a wait operation outcome then it must be executed since its direct predecessor is executed. If Q is a wait operation outcome then the wait operation must be executed and one of its outcomes must be executed. The executed outcome must be able to reach operation I, since otherwise, I would not be executed. Since the executed wait operation outcome can reach I and I has one outcome of the wait operation as a predecessor the executed outcome must be a predecessor of I. Thus, there must be an executed operation that is a predecessor of I and that has J as a direct predecessor. We can call this operation K_1 and if K_1 is not a direct predecessor of I we can repeat the above argument to find an executed operation, K_2, that is a predecessor of I and that has K_1 as a direct predecessor. Since there are only a finite number of operations in the process, an operation, K_n, must eventually be found that is a direct predecessor of operation I. Then since each of the K_i operations is executed and is executed only after its direct predecessors are completed, operation I must be initiated only after operation J is completed. Since operation J is any arbitrary predecessor of I the proof is complete.

The addition of the wait operations to the precedence relation in the manner described above is not necessary to insure an error-free process as Figure 37 shows. This is an alternative to the one in Figure 36 and allows operations 2 and 5 to be executed concurrently if operation 4 is not executed. Executing operations 2 and 5 concurrently may result in sequencing errors but this cannot be determined from the precedence relation since we consider the precedence relation to be transitive. Additional tests would be required to determine whether operations 2 and 5 can be executed concurrently without sequencing errors. Since attempting to identify these cases and allow the additional concurrent operation would complicate the methods described, we will not consider this possibility further.

3.3.5 Proof That The Synthesized Network Is Error-Free

Theorem 3: A concurrent process synthesized from a precedence relation and dominance relation of a loop-free sequential process by adding wait operations to make all direct predecessors of each operation subordinate to the operation and by using cut sets to synthesize the initiation signals for each operation is error-free.

Proof: Each operation is initiated only after its direct predecessors are completed and by Theorem 2 this insures that all predecessors of the operation are completed before it is initiated. Thus, there can be no sequencing errors in the process. Every operation has a

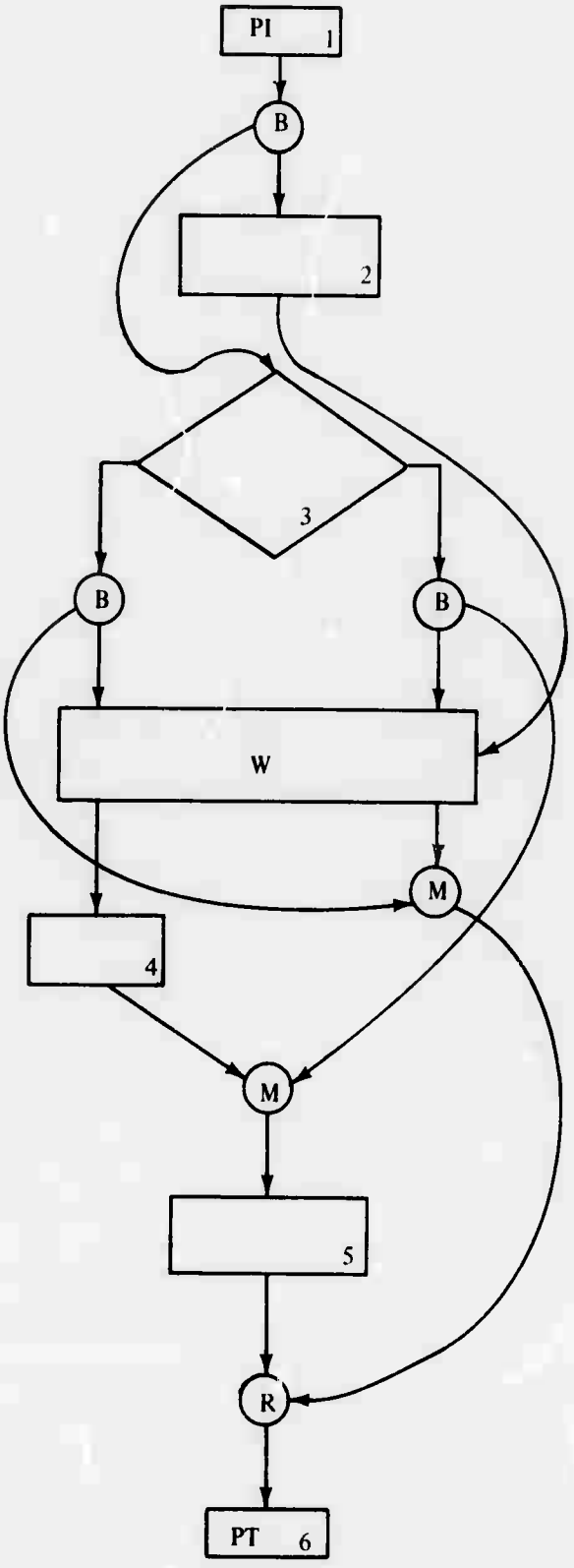


FIGURE 37. A CONCURRENT PROCESS WHERE AN OPERATION DEPENDS ON ONE BUT NOT ALL OUTPUTS OF THE WAIT OPERATION

predecessor since the PI operation is a predecessor of every other operation.The use of cut sets insures that if a rendezvous operation receives an initiation signal on any input it will receive exactly one signal on all inputs and that no merge operation can receive more than one initiation signal so there are no hazard or incomplete rendezvous errors. The completion signal from every operation in the process must be generated before the termination operation is initiated since it has all the operations of the process as predecessors, thus all operations are completed when the termination operation is executed.

3.4 SUMMARY

A summary of the steps used to synthesize a concurrent process from a sequential one, as described in this chapter, is given below.

1. Determine the precedence relation, dominance relation and control graph for the process.
2. Find an operation, I, whose direct predecessors have had their initiation signals synthesized or else whose only direct predecessor is the PI operation.
3. If any direct predecessor of I is not dominated by I add a wait operation and update the precedence relation as described in Sections 3.3.3 and 3.3.4 and then return to step 2. If all direct predecessors of I are dominated by I go to step 4.
4. If any direct predecessors of I have the same dominance relation, connect their outputs to a rendezvous operation and use the rendezvous output in place of the operation outputs.
5. Find the cut sets for operation I and use a cut set table to pick the combinations of signals to be merged together. Rendezvous the output signals from each merge and use the rendezvous output as the initiation signal for I.
6. If I is not the PT operation return to step 2.

4. PROCESSES CONTAINING LOOPS

In a loop-free process no operation can be executed more than once for a single initiation of the process. In terms of the connectivity graph, a loop-free process is one in which there is no chain of arcs from a node that reaches the node again, or the main diagonal of the reachability matrix is all zeroes. Processes containing loops are processes in which some operations may be executed more than once for a single initiation of the process. An example of a process containing a loop is shown in Figure 1. This process calculates the product of the contents of register B and the contents of the right half of double length register A and leaves the result in register A. Operation 5 is a decision which determines whether operations 4 to 8 will be repeated.

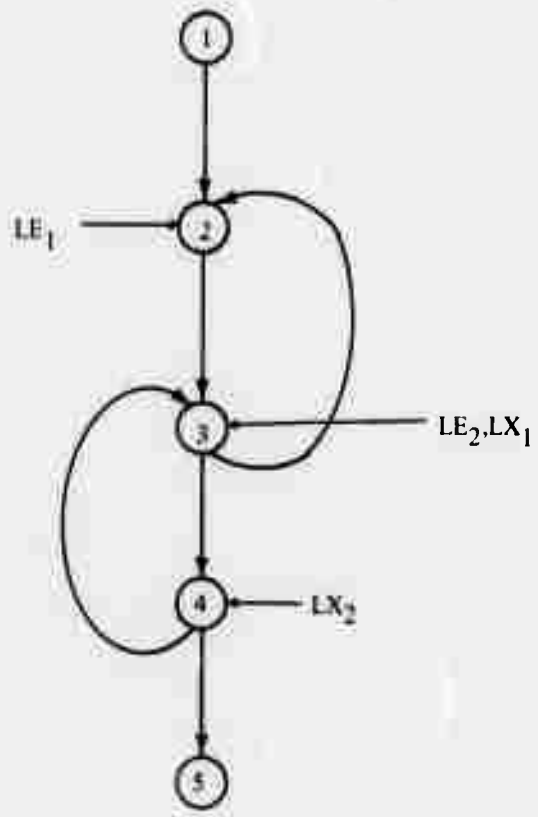
Consider a sequence graph that represents a loop-free process. If the addition of an arc from node I to node J would introduce a loop into the process the arc is called a *feedback arc*. The *loop* corresponding to the feedback arc is the set of nodes that are in the maximally strongly connected subgraph[34] formed by the addition of the feedback arc. Node J is called the *loop entry node* (LE node) of the loop and node I is called the *loop exit node* (LX node). If any node in the loop, except J, has an arc directed to it from a node outside the loop, the node is called a *secondary entry node*. Similarly, a node other than node I that has an arc directed from it to a node outside the loop is called a *secondary exit node*. If two loops have one or more nodes in common but all nodes of one loop are not contained in the other loop, they are called *intersecting loops*. If all nodes of one loop are contained in another loop they are called *nested loops* and if two loops have no nodes in common they are called *disjoint loops*.

The feedback arcs can be determined by calculating M.S.C. subgraphs as described in Section 2.3 and each feedback arc can be identified with two nodes designated as the *loop initiation node* (LI node) and *loop termination node* (LT node) as shown in Figures 38 and 39. These nodes are added to the graph and differ from the LE and LX nodes in that they do not represent operations of the original process. Consider the sequential execution of the process shown in Figure 39. Each time one of the LT nodes is reached the corresponding LI node will be executed next. A possible sequence of operation executions for this system would be:

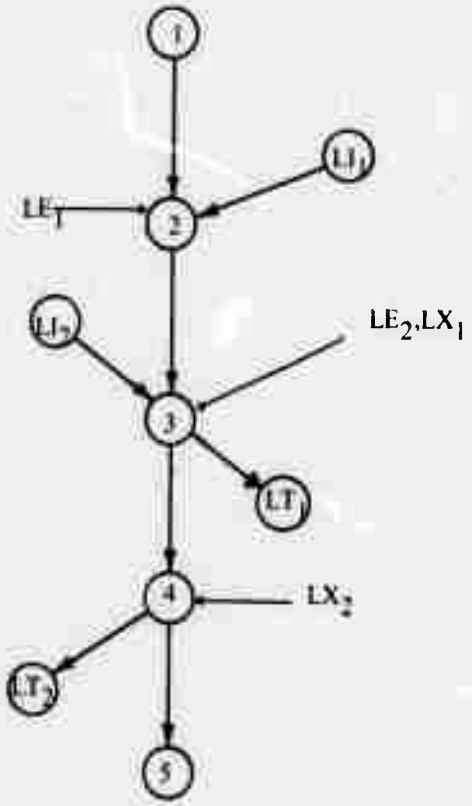
$$1, 2, 3, LT_1, LI_1, 2, 3, LT_1, LI_1, 2, 3, 4, LT_2, LI_2, 3, LT_1, LI_1, 2, 3, 4, 5$$

We will call the execution of the operations from and including an initiation node to and including the following termination node the execution of a *cycle* of the system. Between the execution of the Nth and Mth cycles of a system exactly M - N feedback arcs must be traversed and the first node of the first cycle must always be the PI node while the last node of the last cycle must always be the PT node. The sequence of operation executions given above contains the following five cycles:

cycle 1 $1, 2, 3, LT_1$
cycle 2 $LI_1, 2, 3, LT_1$
cycle 3 $LI_1, 2, 3, 4, LT_2$
cycle 4 $LI_2, 3, LT_1$
cycle 5 $LI_1, 2, 3, 4, 5$



A CONNECTIVITY GRAPH
CONTAINING 2 FEEDBACK ARCS
FIGURE 38



THE CONNECTIVITY GRAPH OF FIGURE 2 WITH THE
FEEDBACK ARCS REPLACED BY LOOP INITIATION
AND TERMINATION NODES
FIGURE 39.

Since an operation may be executed more than once we must distinguish between different executions of the same operation. In the above example operation 3 is executed five times, once for each cycle. A particular execution of operation 3 can be identified by specifying the cycle on which it occurs.

The following section discusses several different types of concurrent operation that are possible in systems containing loops and the next three sections discuss dominance relations, execution relations, and cut sets in processes containing loops. Sections 4.5, 4.6, and 4.7 discuss methods for synthesizing concurrent processes from sequential processes containing loops.

4.1 CONCURRENT OPERATIONS IN PROCESSES CONTAINING LOOPS

Concurrent execution of the operations of a process can be separated into three categories. The ones chosen, although somewhat arbitrary, are convenient because they correspond closely to the types of concurrent execution allowed by the synthesis methods to be presented. In the previous example, concurrent execution of operations 2 and 3 in the same cycle (for example, cycle 1) would be type A concurrency.

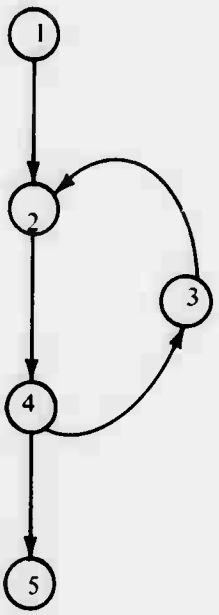
Type B concurrency is concurrent execution of two operations which are in different cycles but where the operation which would be executed first in the sequential execution of the process can reach the other operation even with all feedback ares deleted. In the preceding example, execution of operation 2 of cycle 1 concurrently with operation 4 of cycle 3 would represent type B concurrency while execution of operation 3 of cycle 3 concurrently with operation 4 of cycle 3 would represent type A concurrency.

Finally, type C concurrently represents concurrent execution of two operations where the one that would be executed first in the sequential process can *only* reach the other operation over paths containing one or more feedback ares. An example of type C concurrency would be the concurrent execution of operation 3 of cycle 2 with operation 2 of cycle 3.

4.2 DOMINANCE RELATIONS IN PROCESSES CONTAINING LOOPS

As in the loop-free case, we will require the dominance relation between each operation and its direct predecessors. The introduction of loops complicates the determination of dominance relations since some operations may be executed more than once for a single initiation of the process. Consider the example shown in Figure 40 where operations 2, 3, and 4 may be executed on any number of cycles but operations 1 and 5 will be executed only once for each initiation of the process. If operation 2 is executed, operation 1 must be executed also, however, operation 2 may be executed many times while operation 1 is executed only once. In this case only the execution of operation 2 on the first cycle corresponds to the execution of operation 5. Therefore operation 1 does not dominate operation 2, and operation 5 does not dominate operation 4. The definition of dominance given in Section 2.5 thus must be revised to the following:

> Operation I dominates operation J if for *every* execution of operation J, operation I must be executed also.



CONNECTIVITY GRAPH

	1	2	3	4	5
1	1	0	0	0	1
2	1	1	1	1	1
3	0	0	1	0	0
4	1	1	1	1	1
5	1	0	0	0	1

DOMINANCE RELATION

FIGURE 40. A CONNECTIVITY GRAPH AND ITS DOMINANCE RELATION

The dominance relation in Figure 40 was developed from the connectivity graph by inspection. It shows that the conditions for the execution of operations 1 and 5 and of 2 and 4 are equal and that operations 2 and 4 dominate all operations of the process.

If any operation, J, is a direct predecessor of an operation, I, then the dominance relation between them must be determined. First consider the case where operation J can reach operation I only over paths which contain a feedback arc. Then if J is executed, I may or may not be executed since there will be a path from J to the PT node which does not include I. Therefore I will not dominate J. Operation J may dominate operation I but this information is not required for the synthesis since the only information required between two operations is whether they dominate each other, and whether an operation dominates its predecessor. Both of these questions are answered by the fact that operation I cannot dominate operation J.

Next consider the case where operation J can reach operation I over a path that contains no feedback ares. In Section 2.5 the dominance relation for a loop-free process was found by deleting each node in succession and determining which remaining nodes could not reach the Nth node or be reached from the first node of the connectivity graph. Any node that could not reach the Nth node or be reached from the first node was dominated by the deleted node. In the example of Figure 40, deletion of node 1 prevents any other operation from being reached by the first node. However, node 1 does not dominate all of the other nodes because of the loop. Operations 2, 3, and 4 can be initiated on a path other than the one starting at the PI node, specifically the one initiated by the feedback arc.

For a system with loops the calculation of the dominance relation described in Section 2.5 must be revised slightly. Instead of merely testing whether an operation can reach the Nth node and be reached by the first node, a test must also be made to determine whether the operation can reach itself. For if operation J can reach itself after node I has been deleted, operation J can be executed any number of times without a corresponding execution of operation I. Thus operation I dominates J if deletion of operation I prevents node J either from reaching the Nth node or being reached from he first node, and prevents operation J from reaching itself. Otherwise operation I does not dominate operation J.

Theorem 4: Node I dominates node J if and only if deletion of node I prevents node J either from reaching the Nth node or being reached by the first node of the connectivity graph and prevents J from reaching itself.

Proof: Consider any two operations, I and J, and assume that operation I can reach operation J over a path containing no feedback ares. The first time that J is executed it must be reached by a path from the first node that does not include operation J. The following executions of operation J require a path from operation J to itself. If deletion of operation I prevents operation J from being reached by itself or the PI operation, then operation I must be on every path from the PI operation to operation J and every path from operation J to itself. Then operation I must be executed every time that operation J is executed and, therefore, operation I dominates operation J. Next assume that operation I dominates operation J or that operation I must be executed every time that operation J is executed. Then operation I must be on every path from the PI operation to

operation J, since otherwise operation J could be executed without a corresponding execution of operation I. Similarly, operation I must lie on every path from J that reaches J again or there could be an execution of operation J without a corresponding execution of operation I. Thus, when J follows I, operation I dominates operation J if, and only if, deletion of operation I removes all paths from the PI operation to J and from J to itself. Similar arguments can be used to show that when I follows J, operation I dominates operation J if and only if deletion of operation I eliminates all paths by which J can reach the PT operation or itself.

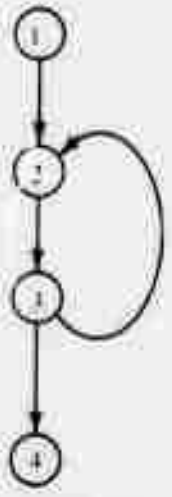
4.3 PRECEDENCE RELATIONS IN PROCESSES CONTAINING LOOPS

A precedence relation cannot be established directly for a graph containing loops[50] since attempting to do so will usually result in an operation being required to precede itself. This is illustrated in the example of Figure 41. If operation 2 must be completed before operation 3 can be initiated and operation 3 must be completed before operation 2 is initiated the second time, the precedence diagram is as shown in Figure 41, which requires operation 2 to be completed before it can be initiated. This occurs because the precedence relation makes no distinction between the first and second time that operation 2 is executed. One solution, suggested by Marimont[50], is to "unwind" the loop as shown in Figure 42. Each successive iteration of the loop is represented by repeating the nodes that form the loop. This transforms the cyclic graph into a loop-free graph and allows the precedence relations to be established if the number of cycles is known.

Another method[9, 36, 37], which will be used here, is to repeat one copy of the process and to establish the precedence relation between operations in the two copies. The repeated copy of the process is used to allow the establishment of the precedence relation between operations which can be reached only over paths containing feedback arcs. This is illustrated in Figure 43 where the nodes with unprimed numbers represent operations in one copy of the process and the nodes with primed numbers represent the operations from the preceding copy. Operation 2 follows the process initiation node or operation 3 of the preceding cycle, depending on whether operation 2 is being executed for the first time. In the example shown in Figure 44, operation 2 and operation 3 of the first cycle may be executed concurrently, and operation 6 may be executed concurrently with all cycles of the process. Also, operation 4 of one cycle may be executed concurrently with operation 3 of the preceding cycle.

The method described in Section 2.4 for determining the decision precedence relation must be modified slightly when a process contains loops. In the connectivity graph of Figure 45 operations 5 and 6 do not require decision operation 4 to be completed before they are initiated while in Figure 46, operations 5 and 6 cannot be initiated until decision operation 4 is executed and the decision outcome is to execute operation 5. Operations 5 and 6 must follow decision operation 4 in Figure 46 because there is a path from the decision to the termination node that does not pass through operations 5 and 6. Thus, until the outcome of decision 4 that connects to operation 5 is taken, it is possible that operations 5 and 6 will not be executed. In the example of Figure 45 we exclude the possibility of the loop being executed an infinite number of times and therefore operations 5 and 6 must be executed eventually, regardless of the outcome of any particular execution of the decision.

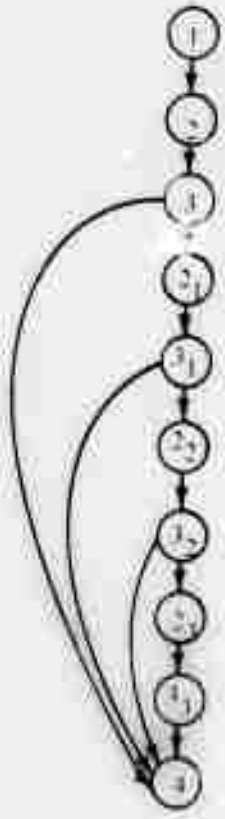
Consider any possible sequential execution of a process up to and including the execution of operation I. I must depend on the closest decision, J, to I that can reach the PT node without passing through I, since until that decision



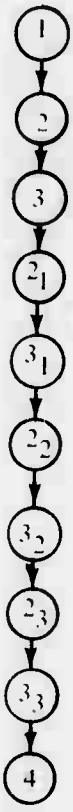
CONNECTIVITY GRAPH

PRECEDENCE GRAPH

FIGURE 41. A PROCESS WITH A LOOP

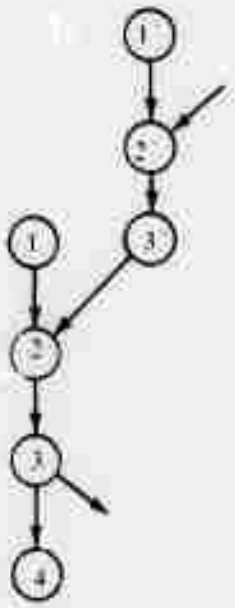


SEQUENCE GRAPH

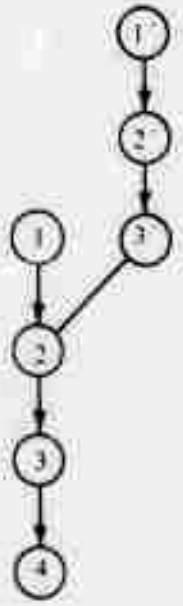


PRECEDENCE GRAPH

FIGURE 42. THE PROCESS OF FIGURE 41 WITH THE LOOP "UNWOUND"



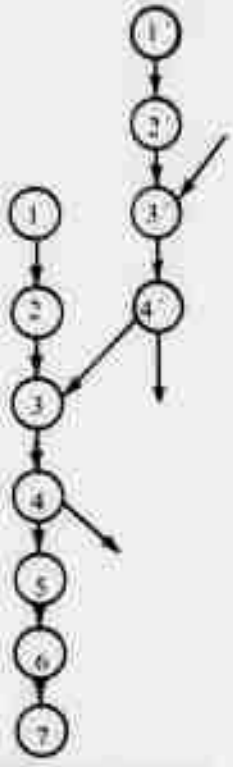
CONNECTIVITY GRAPH



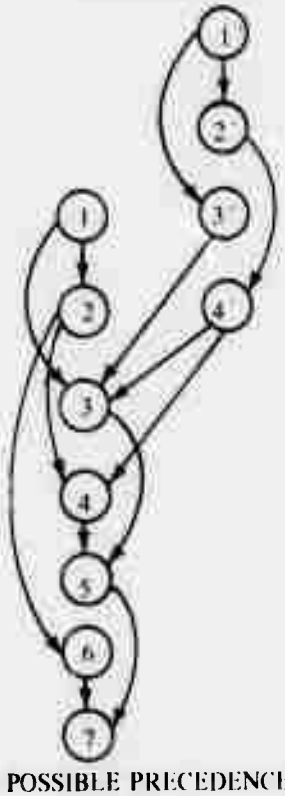
PRECEDENCE GRAPH

FIGURE 43. THE PROCESS OF FIGURE 41 WITH ONE COPY OF THE LOOP REPEATED

CONNECTIVITY GRAPH



THE CONNECTIVITY GRAPH WITH A REPEATED COPY OF THE PROCESS



A POSSIBLE PRECEDENCE GRAPH

FIGURE 44. A PROCESS WHICH ALLOWS CYCLE TO CYCLE CONCURRENCY

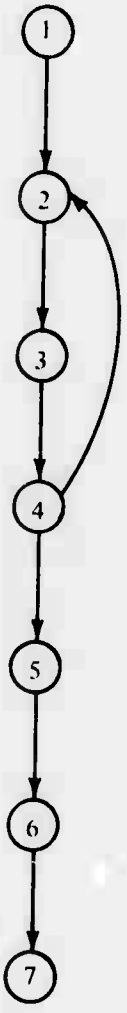


FIGURE 45
A Process in Which the Execution of Operation 5 may Preceed the Execution of operation 4

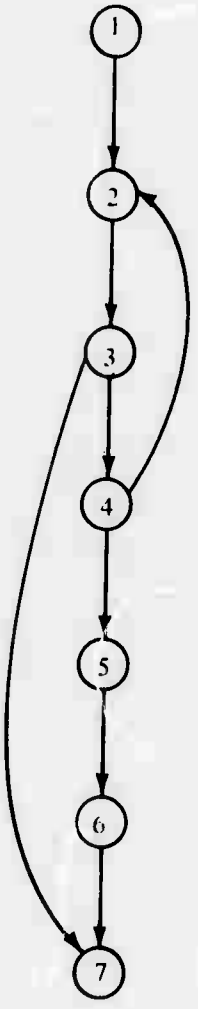


FIGURE 46
A Process in which the Execution of Operation 5 may not preceed the Execution of operation 4

is executed I may be bypassed. The outcome of the decision that reaches I must be dominated by I since if that outcome is taken, I must be executed because no path can bypass I and because decision J cannot be reached again before I is executed. Thus, if a decision has an outcome that is dominated by operation I and the decision can reach the PT node with I deleted, then operation I must have that decision as a predecessor. I cannot have any other decision along the same path as a direct predecessor since when decision J is completed, operation I can be initiated, without considering the outcome of any other decisions. Thus, I can have as direct predecessors only those decisions which can reach the PT node with I deleted and that have an outcome that is dominated by I. Conversely any decision that satisfies these two requirements will be a direct predecessor of I for some particular execution sequence. The required information can easily be found and the decision precedence relation determined when the dominance relation is calculated.

If the loops in a process are nested, repeating one copy of the process will allow the precedence relation between any pair of operations to be determined[36, 37]. However, if the loops are intersecting, as shown in the example of Figure 47, the members of the precedence relation between some operations may not be found. In this example, assume that operation 3 is required to follow operation 6 when the operations are executed in the following order: . . 6, 7, 4, 5, 2, 3, . . . This cannot be determined from the precedence relation found from a single repetition of the process since the path from operation 6 to operation 3 is not included. In fact, if any operation, I, can be reached from another operation, J, only over a path which includes two or more feedback arcs, the path will not be shown explicitly when only two copies of the process are used. Additional copies of the process can be used to make all simple paths between operations explicit, but with a large number of feedback arcs many copies of the process could be required. In order to require consideration of only one additional copy of the process, members will be added to the precedence relation to require that operation J precede operation I when there is a path from J to I that contains two or more feedback arcs and no path from J to I that contains only one of the feedback arcs.

Consider a process with a path from an operation, J, to an operation, I, containing two feedback arcs such that no path from J to I contains only one of the feedback arcs and let the first feedback arc traversed be from LX_1 to LE_1 and the second one from LX_2 to LE_2. Then the following conditions must be satisfied by the feedback free connectivity graph where $L \rightarrow M$ indicates that L reaches K and $I \not\rightarrow K$ indicates that L does not reach K.

1) $J \rightarrow LX_1$
2) $LE_1 \rightarrow LX_2$
3) $LE_2 \rightarrow I$
4) $LE_1 \not\rightarrow I$
5) $J \not\rightarrow LX_2$

The first three conditions are necessary for the path from J to I to exist. The fourth condition must be true or the following path, containing only the first feedback arc would exist:

$$J, \ldots, LT_1, LI_1, \ldots, I$$

Similarly if the fifth condition is not satisfied a path from J to I will exist that contains only the second feedback arc.

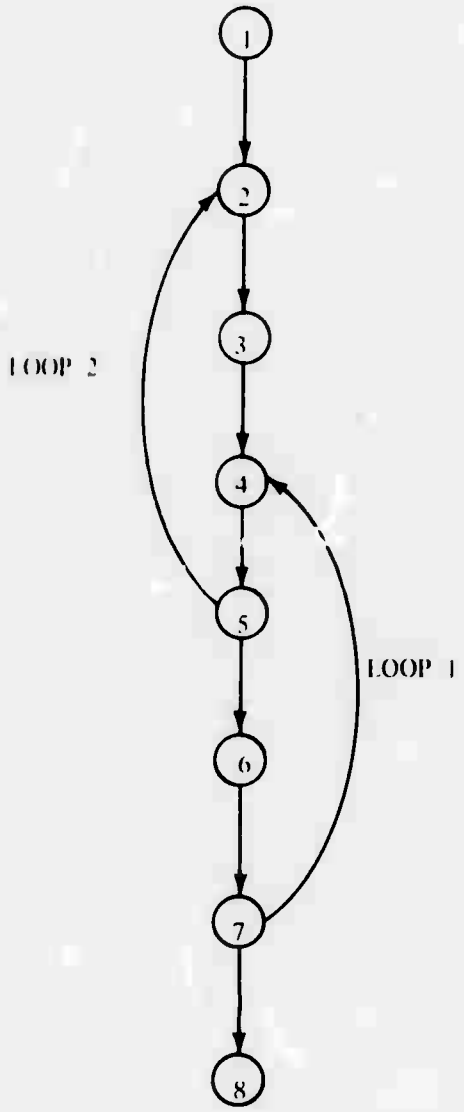


FIGURE 47. A CONNECTIVITY GRAPH IN WHICH OPERATION 6 CAN REACH OPERATION 3 ONLY OVER A PATH CONTAINING 2 FEEDBACK ARCS

The following three conditions can be derived from conditions 1 through 5 and the fact that each LE node must reach the corresponding LX node.

6) $LX_1 \not\rightarrow LE_2$
7) $LE_1 \not\rightarrow LE_2$
8) $LX_1 \not\rightarrow LX_2$

Condition 6 must be true, for otherwise condition 1 would imply that $J \rightarrow LE_2$ and therefore that $J \rightarrow LX_2$ in contradiction to condition 5. Conditions 7 and 8 must be true for if either one is not true there is a path from J to I that includes only one of the feedback arcs.

Thus, in any case where an operation, J, can reach an operation, I, over a path containing two feedback arcs but not over a path which does not contain both feedback arcs, conditions 2, 6, 7, and 8 must be satisfied. Several examples are shown in Figure 48. In every case operation I will have LX_2 as a predecessor since operation I cannot be executed until the decision outcome specifies that the feedback arc is to be taken. If operation J is made a predecessor of LX_2 then I cannot be executed until J is completed. This can be accomplished by adding a member to the precedence relation from J to LX_2. In general, we can add a precedence relation from any operation, K, to each LX node that K can reach only over a feedback arc and whose corresponding LE node operation K cannot reach over that feedback arc. This will insure the existance of a precedence relation between any operation and another operation which can reach it only over a path containing two feedback arcs. Therefore, two copies of the process will be sufficient to express all members of the direct precedence relation. In the example of Figure 47 this will require operation 6 to be completed before operation 5 of the following cycle is initiated. This reduces the possibilities for concurrent operation since operation 6 of one cycle and operation 5 of a following cycle cannot be executed concurrently even if they are independent, but it simplifies the concurrent process synthesis and the representation of the precedence relation.

4.4 CUT SETS IN PROCESSES CONTAINING LOOPS

As in the loop-free case, cut sets of the control graph will be used to determine which completion signals from the direct predecessors of an operation, I, can be merged together. A cut set must consist of a set of nodes of the control graph such that one and only one node of the set will be in each path from the initiation node to operation I or from operation I to itself. A cut set for an operation, I, is thus defined as a set of nodes of the control graph satisfying the following four conditions:

1. Each node can reach I over a feedback free path.
2. Each node is dominated by I.
3. No member of a cut set can reach any other member over a path that does not include I.
4. If any node is added to the set then one of the first three conditions will be violated.

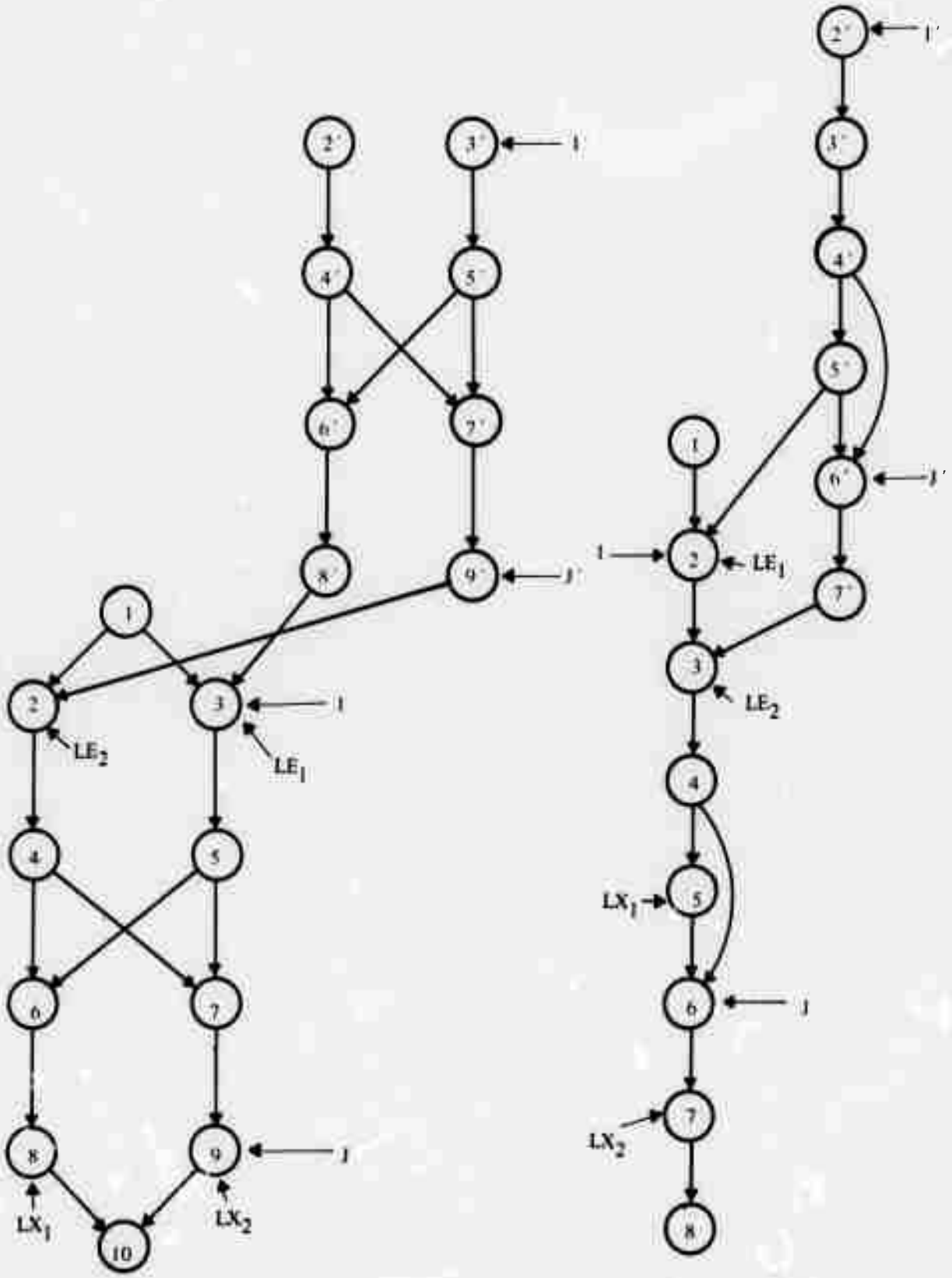


FIGURE 48. TWO EXAMPLES OF PROCESSES WHERE OPERATION J CAN REACH OPERATION I ONLY OVER A PATH CONTAINING TWO FEEDBACK ARCS

Consider the example of a control graph shown in Figure 49. All nodes which may be members of the cut sets for I must satisfy conditions 1 and 2 and these nodes are drawn with a heavier circle. The cut sets are:

$$\{5, 4, 14\}, \{5, 4, 13, 15\}, \{5, 10, 16, 15\}, \{9, 16, 15\}.$$

Clearly every node that reaches I and is dominated by I must be in some cut set for I. Now we must prove that any path from the initiation node to I or from I to itself must pass through a member of each cut set for I.

Theorem 5: Every path in the control graph from the initiation node to I or from I to itself must pass through exactly one node of each cut set.

Proof: No path can pass through more than one node of any cut set since if it did, one node of the cut set would be able to reach another node of the cut set, in contradiction to the rules for forming cut sets. Next we must show that at least one node from each cut set will be on every path. Consider the sequential execution of any path in the control graph reaching I and let it be composed of nodes $N_1, N_2, N_3 \ldots N_i, \ldots N_j \ldots N_k$, I in that order. Let N_i be the last node that reaches any member of a cut set, $Z = \{z_1, z_2, z_3 \ldots z_i\}$. Let N_j be the first node that can be reached by a member of the cut set. Since a member of the cut set can reach N_j and since N_i can reach a member of the cut set, N_i must precede N_j or a member of the cut set would be able to reach some other member. Furthermore, since N_i must have at least two extant arcs, there must be at least one node, N_{j-1}, between N_i and N_j of the control graph. If node N_{j-1} is dominated by I then Z is not a cut set because N_{j-1} could be added to it without violating any of the restrictions. Thus if N_{j-1} is dominated by I, a member of the cut set Z must lie on the path reaching I.

If I does not dominate N_{j-1} then N_{j-1} must be able to reach the PT node or itself, with node I deleted. Node N_{j-1} cannot reach the termination node with I deleted, since if so N_j, and thus a member of the cut set could reach the termination node with I deleted. Then N_{j-1} must be able to reach itself and therefore N_j must be able to reach N_{j-1}. Let $N_{j-\ell}$ be the earliest node that N_j can reach. Node $N_{j-\ell}$ must come after N_i since, if not, a member of the cut set could reach some other member. Also, as before, there must be a node $N_{j-\ell-1}$ between N_i and $N_{j-\ell}$. If $N_{j-\ell-1}$ is dominated by I then Z is not a cut set. If node $N_{j-\ell-1}$ is not dominated by I then it must be able to reach the PT node or itself with node I deleted. If it could reach the PT node the member of the cut set reaching N_j could reach the PT node also, and Z would not be a cut set. If $N_{j-\ell-1}$ cannot reach the PT node it must be able to reach itself, but then N_{j-1} could reach $N_{j-\ell-1}$ and therefore N_j could reach $N_{j-\ell-1}$ in contradiction to the assumption that node $N_{j-\ell}$ was the earliest node that node N_j could reach. Thus, there must be a member of each cut set of I on every path to node I from itself or from the PI node.

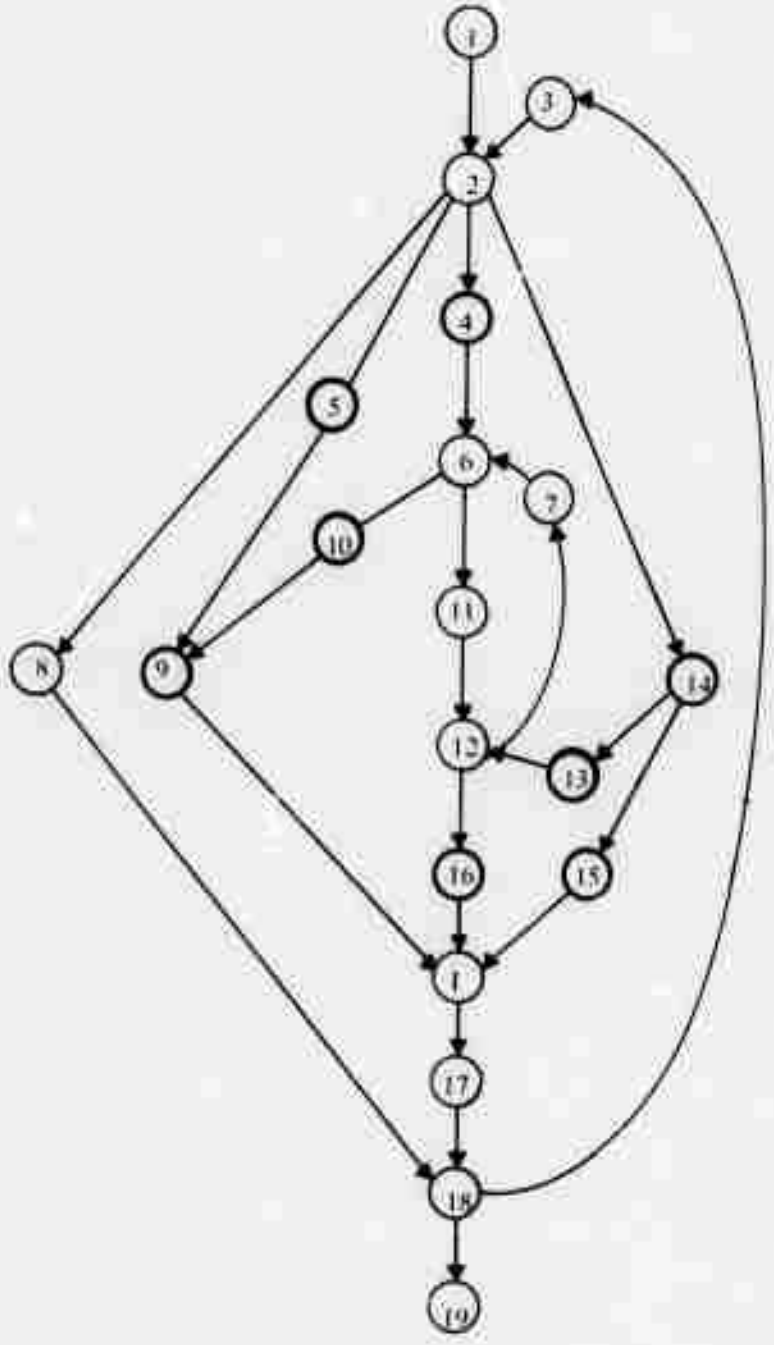


FIGURE 49. A CONTROL GRAPH

4.5 SYNTHESIS OF PROCESSES WITH TYPE A CONCURRENCY

A simple method for treating processes with loops is to add additional members to the precedence relation that will eliminate any concurrency between cycles. That is, we will require all operations in one cycle to be completed before any operations in the following cycle are initiated. When the precedence relation is calculated, a single copy of the process is used, and members are added to the precedence relation to require every operation which reaches an LT node to precede it and to require every operation which can be reached by an LI node to follow it. A decision and merge are added to the process as shown in Figure 50, with an output for each initiation node and an input for each termination node, respectively. The decision represents the possibility of initiating the process by the process initiation node or by one of the loop initiation nodes. These additions transform the process into a loop-free process. The dominance relation and precedence relation for the process can be determined as discussed in Section 2.5 since the modified graph is loop-free.

After the precedence relation and dominance relation are determined a concurrent process is synthesized as in the loop-free case, except that the added decision and merge operation are omitted. The output signal from each LT operation is then connected to the input of the corresponding LI operation to complete the process. Since every operation that can reach an LT node is a predecessor of it, the LT operation cannot be executed until every operation in the cycle of the process which it terminates is completed. Also, no operations in a succeeding cycle can be initiated until the loop initiation node that starts that cycle is completed. Thus, all operations in one cycle of the process are executed before any operations in the following cycle are initiated. The execution of the complete process can be considered to be a series of executions of a loop-free process, with possible concurrent execution of operations within each execution of the loop-free process. There is, however, no possibility of concurrent execution of operations in different cycles.

4.5.1 Example

Figures 51 to 54 show an application of this method to the process for multiplying two 12-bit numbers that is shown in Figure 1. The multiplier is in register B and the multiplicand is originally in AR, the right half of double length register A. Register C is used as a counter to record the number of cycles completed.

Figure 51 shows the connectivity graph, precedence graph, and dominance matrix for the process, with the feedback arc replaced by LI and LT nodes. The flow chart for the process synthesized from these relations is shown in Figure 52. The operations of incrementing and testing the counter are performed concurrently with the operations to test the multiplier, add the multiplicand, and shift the partial product. Also, the initializing operations of clearing registers C and AL are performed concurrently with some other operations in the first cycle.

Several changes can be made to reduce the amount of concurrent operation permitted, and thereby reduce the number of control operations required. The execution of operations 2 and 3 with operations in the first cycle of the loop is called *loop entrance concurrency* and can be eliminated if operations 2 and 3 are required to be completed before any operations within the loop are initiated, thereby eliminating the need for the wait operation that follows decision operation 4. This will reduce the time required for each cycle of the process since there will be fewer control operations in the loop, but will eliminate the concurrent execution of operations 2 and 3 with other

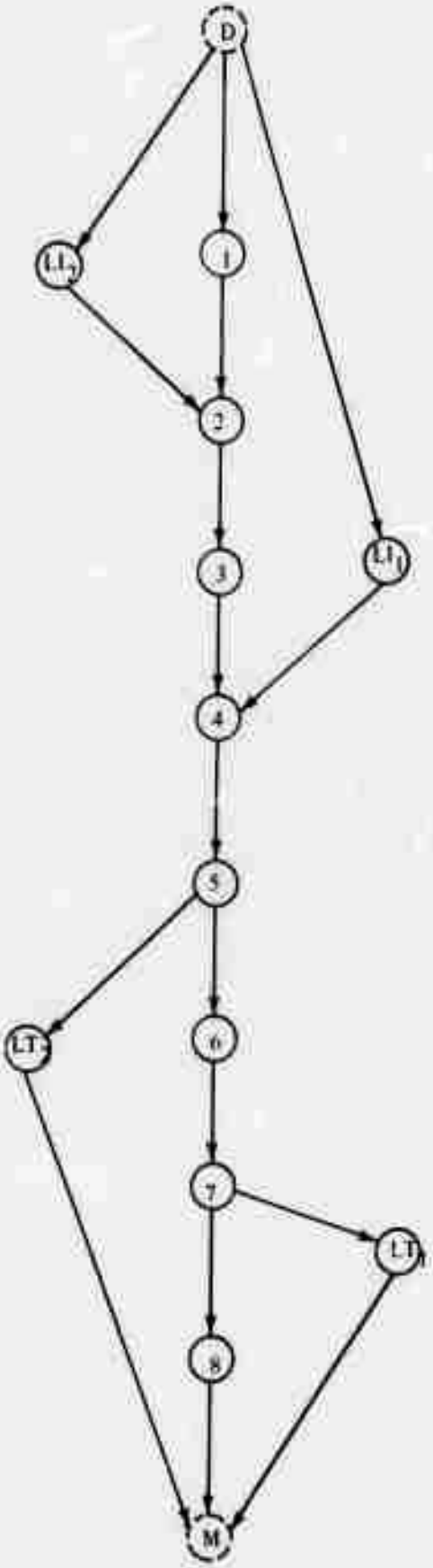


FIGURE 50. A CONNECTIVITY GRAPH WITH ADDED DECISION AND MERGE OPERATION

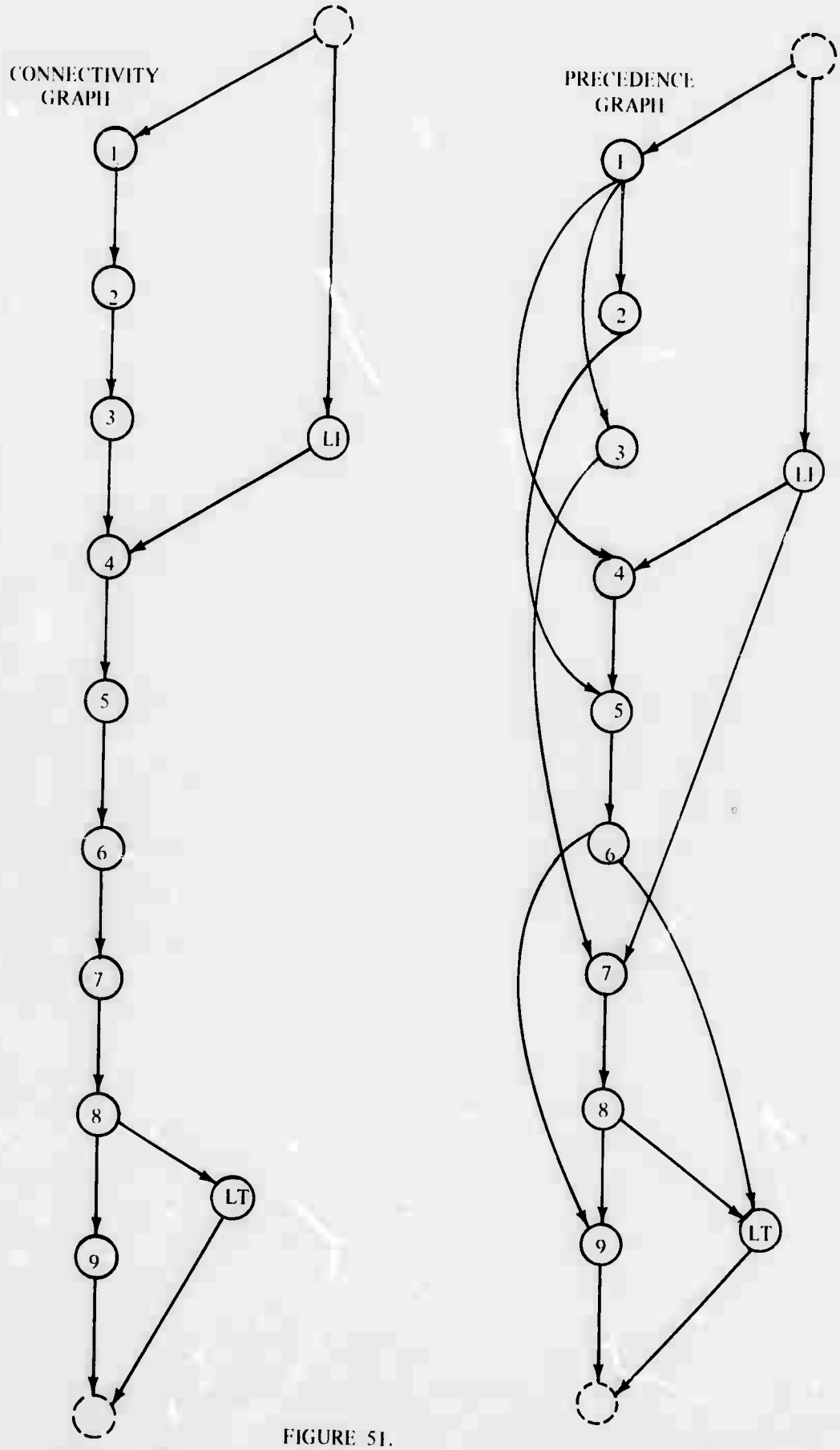


FIGURE 51.

Connectivity graph, Precedence graph, and Dominance Relation for the Example of Figure 1

	1	2	3	4	5	6	7	8	9	LI_1	LT_1
1	1	1	1	0	0	0	0	0	0	0	0
2	1	1	1	0	0	0	0	0	0	0	0
3	1	1	1	0	0	0	0	0	0	0	0
4	1	1	1	1	1	1	1	1	1	1	1
5	0	0	0	0	1	0	0	0	0	0	0
6	1	1	1	1	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1	1	1	1	1
8	1	1	1	1	1	1	1	1	1	1	1
9	0	0	0	0	0	0	0	0	1	0	0
LI_1	0	0	0	0	0	0	0	0	0	1	0
LT_1	0	0	0	0	0	0	0	0	0	0	1

FIGURE 51.

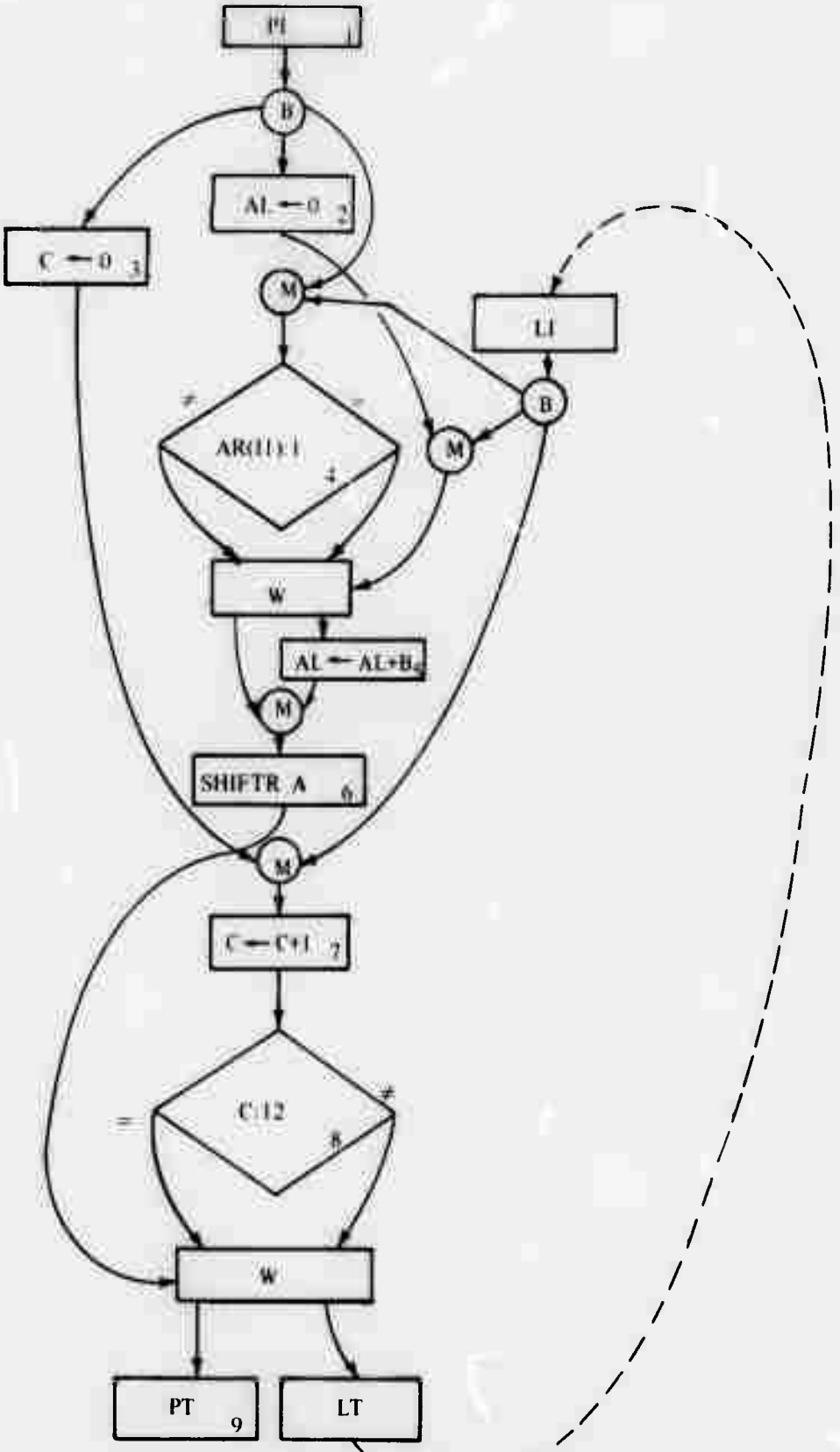


FIGURE 52. CONCURRENT FLOW CHART FOR THE EXAMPLE OF FIGURE 1

operations of the first cycle. Whether the overall result will be an increase or decrease in the total execution time depends on the number of cycles that will be executed and the ratio of time saved in each cycle of the process to the additional time required for initialization of the first cycle.

To determine the control network for the process with no entrance concurrency, an additional node, X, can be added to the precedence graph and connectivity graph as shown in Figure 53. Every operation of the loop is required to follow the added node and every operation outside the loop which is a predecessor of an operation within the loop is made a predecessor of the added node. Figure 53 also shows the flow chart for the concurrent process synthesized with the added operation. Comparison of Figures 52 and 53 shows that the delay associated with one wait operation has been eliminated from the execution of each cycle.

Another step which can be taken to reduce the control network complexity is to make every operation in the loop precede decision 8, thereby eliminating concurrent execution of operations with the loop exit decision. This will eliminate the need for the wait operation connected to the output of decision 8, but will increase the operating time since decision 8 will not be executed concurrently with any other operations. In this case the increase in operating time will be slight since the time required for the decision is small. In a similar situation the decision might be a more complex operation which required considerable time to complete, and in that case the increase in operating time would be much greater. The precedence graph, with all operations required to be completed before operation 8, and the corresponding control flow chart are shown in Figure 54. Table 1 gives a comparison of the operating times and module requirements for the macromodular implementation of the different configurations. For comparison the times and module requirements are also given for the sequential system of Figure 1 and for a system with all operations completed before decision 8 but with entrance concurrency. The times assumed for the various operations are listed in Table 2. The calculations for total times assume that the addition (operation 5) is performed for half of the cycles executed.

Several interesting observations can be made from the table. First, significant operating time savings can be achieved by allowing concurrent operation. If α is equal to 200ns, one of the concurrent systems requires only 64% of the time required for the completely sequential system. Second, although more equipment is required for the system with concurrent operations, the number of multiplications per second per module can be increased by using concurrent operations in this example. Finally, the system with the most concurrent operation does *not* have the shortest operating time. The system with entrance concurrency is slower than the one without this concurrency because the additional control operations required increase the execution time slightly for each cycle of the process, while speeding up only the first cycle.

4.6 SYNTHESIS OF PROCESSES WITH TYPE B CONCURRENCY

The method developed in the preceding section is effective for some processes but may severely restrict the concurrent execution of operations in processes like the example shown in Figure 55. Since the method of Section 4.5 requires all operations in one cycle to be completed before any operations in the following cycle can be initiated, the only possible concurrent execution of operations in the two loops would occur on their only common cycle, the last cycle on which loop 1 is executed and the first cycle on which loop 2 is executed.

In order to allow concurrent execution of operations in different cycles we will add precedence relations from the LI nodes to all nodes that can be reached by the LI node and cannot be reached by the corresponding LX node and

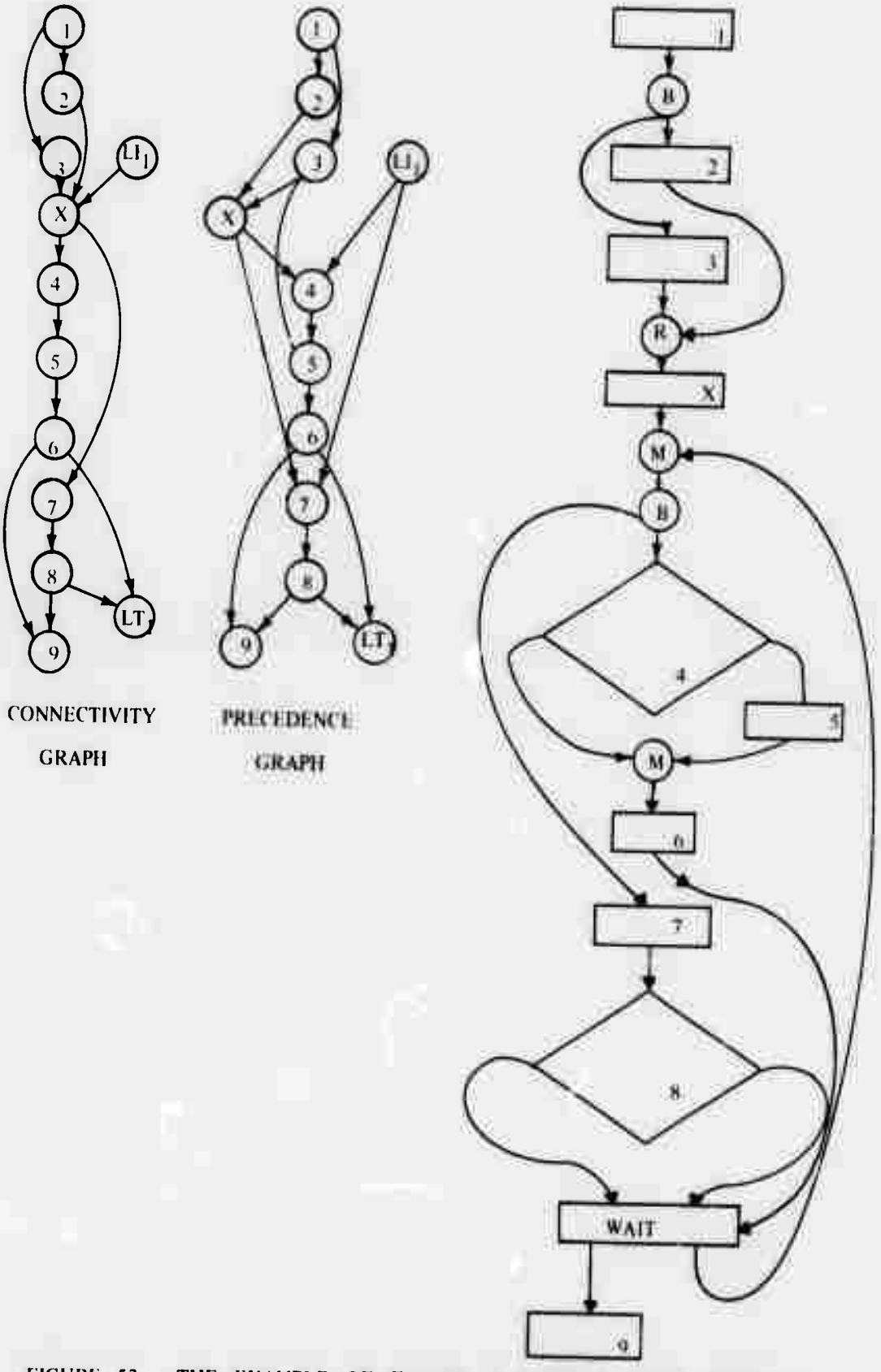
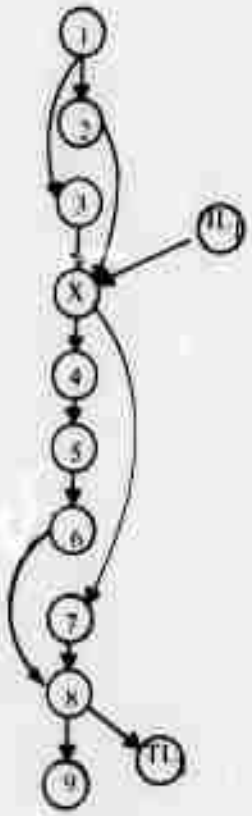


FIGURE 53. THE EXAMPLE OF FIGURE 52 WITHOUT LOOP ENTRANCE CONCURRENCY



PRECEDENCE GRAPH

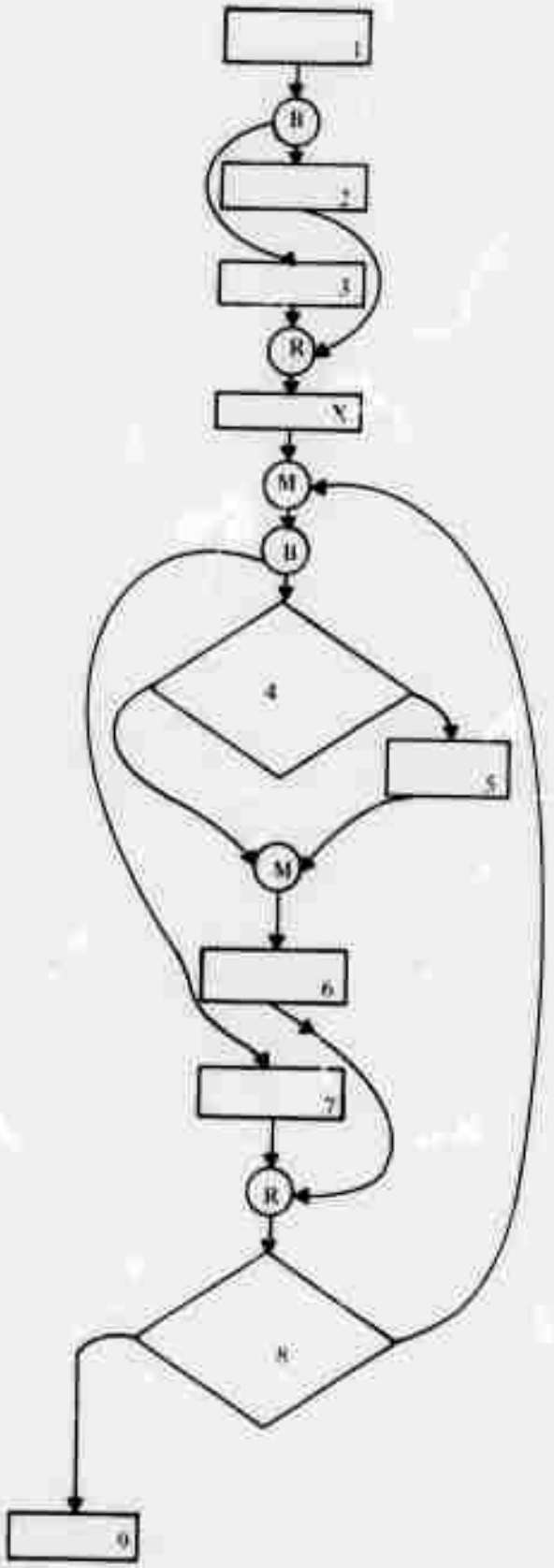


FIGURE 54. EXAMPLE OF FIGURE 52 WITH ALL OPERATIONS PRECEEDING OPERATION 8

COMPARISON OF EXECUTION TIMES AND MODULE REQUIREMENTS

System	Figure	Time* in ns For One Cycle If the Addition is Performed	Total Time in ns Including Initialization	Time in ns (α = 200)	Execution Time Relative To The Sequential System (α = 200)	Total Modules Required	Multiplications Pers Second Per Module (α = 200)
Maximum Concurrency	52	195 + 2α	2220 + 19α	6020	0.65	13	$1.28 \cdot 10^4$
No Entrance Concurrency	53	175 + 2α	2135 + 19α	5935	0.64	12.1	$1.38 \cdot 10^4$
No Exit Decision or Entrance Concurrency	54	275 + 2α	3335 + 19α	7135	0.77	11.75	$1.19 \cdot 10^4$
No Exit Decision Concurrency		295 + 2α	3435 + 19α	7235	0.78	12.6	$1.09 \cdot 10^4$
Sequential		240 + 3α	2880 + 32α	9280	1.00	11.25	$0.97 \cdot 10^4$

*α = Time To Perform One Processing Operation Such As Add, Shift, etc.

TABLE 2

EXECUTION TIME FOR MACROMODULE OPERATIONS

Operation	Time	
Decision	100 ns	
Merge	20 ns	
Branch	15 ns	
Rendezvous	20 ns	
Wait	20 ns	From the time the last of the two initiation Signals is received
Add, Shift, Etc.	α ns	

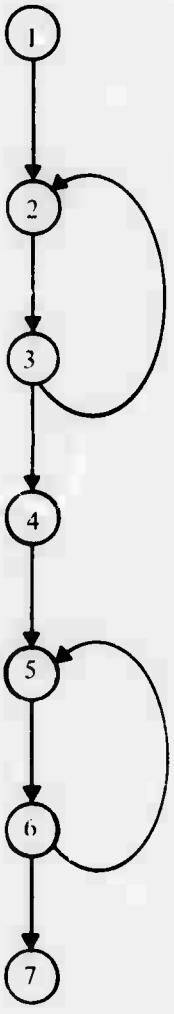


FIGURE 55. A CONNECTIVITY GRAPH WITH TWO LOOPS

to each LT node from operations that can reach it and cannot reach the corresponding LE node. These precedence relations will insure that if any operation, J, can reach another operation, I, only over a path that contains a feedback arc, that operation J will be completed before operation I is initiated. If operation J can reach operation I over a path that does not contain a feedback arc then the precedence graph will show any required precedence relations. When the precedence relation and dominance relation are determined the flow chart for an equivalent concurrent process can be synthesized as follows:

1. Determine the dominance relation as described in Section 4.2.
2. Determine the precedence relation for a single copy of the process requiring each operation to be a predecessor of any LT node it reaches if it does not reach the corresponding LE node and requiring each LI node to be a predecessor of any node not reached by the corresponding LX node.
3. Select an operation, I, other than an LI or the PI operation, whose direct predecessors all have either had their M–R networks synthesized or else are LI or PI nodes.
4. If any direct predecessor, K, of I is not subordinate to I, find the closest decision to K that can be reached by K, that reaches I, and that has an output that is dominated by I. Add a wait operation that receives its decision inputs from the decision and make K a predecessor of the wait operation and the wait operation a predecessor of operation I. Update the direct precedence relation as described in Section 3.3.4 and return to step 3.
5. If any direct predecessors of I have the same dominance relation, connect their outputs to the inputs of a rendezvous operation and use the rendezvous output in place of the operation outputs.
6. Determine the cut sets for I and use a cut set table to choose a group of cut sets that cover all of the predecessors of I and synthesize the corresponding M–R network.
7. If all operations have not had their M–R networks synthesized return to step 3.
8. Connect the loops by combining each LT node with the corresponding LI node.

Unfortunately, the author has not been able to prove that a process synthesized as described above will be error-free, however, no counter-examples have been found. Figure 56 shows a connectivity graph, a precedence relation and the concurrent process that would be synthesized from them. The two loops of the process can be executed independently and type B concurrency is allowed since operations in any cycle of the second loop may be executed before the execution of the first loop is completed.

4.7 SYNTHESIS OF PROCESSES WITH TYPE C CONCURRENCY

The methods developed in the preceding sections are effective for some systems but do not allow any concurrent operation between operations in successive cycles of the same loop. In order to consider concurrent operations between successive cycles of a loop the precedence relation between operations in two copies of the process must be determined as discussed in Section 4.2.

Figure 57 shows an example of a system with a single loop and the corresponding precedence and connectivity graphs including a repeated copy of the loop. When the initiation network for each operation is synthesized, the

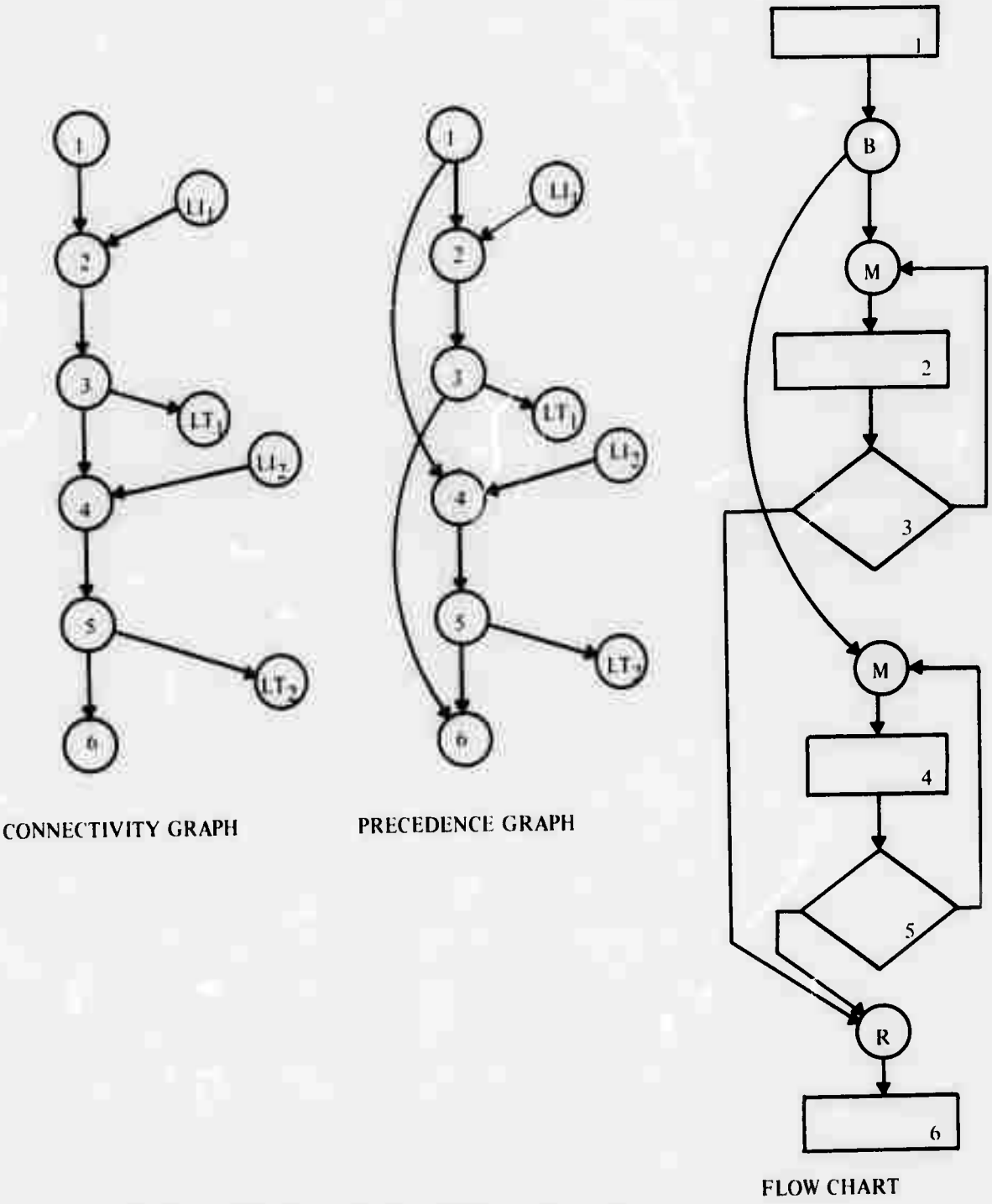
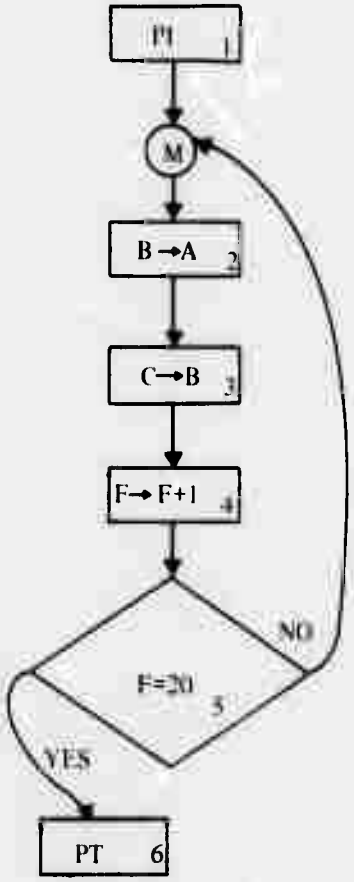
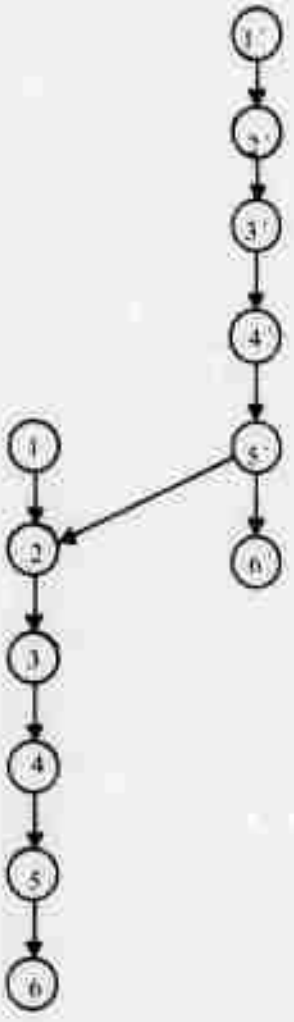


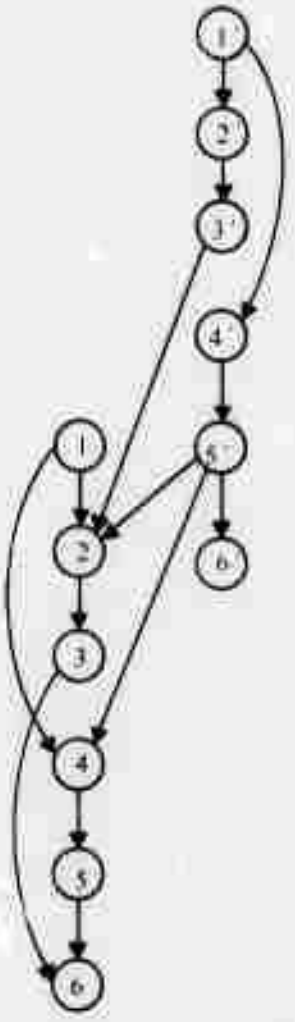
FIGURE 56. AN EXAMPLE OF TYPE B CONCURRENCY



CONTROL FLOW CHART



CONNECTIVITY GRAPH



PRECEDENCE GRAPH

FIGURE 57. A PROCESS WITH A SINGLE LOOP

	1	2	3		5	6
1	1	0	0	0	0	1
2	1	1	1	1	1	1
3	1	1	1	1	1	1
4	1	1	1	1	1	1
5	1	1	1	1	1	1
6	1	0	0	0	0	1

DOMINANCE MATRIX

primed nodes are considered to represent distinct operations although they actually represent the same operations as the unprimed nodes for a preceding cycle. Initiation networks will be required only for the unprimed nodes since they include all operations of the system.

The initiation networks which would be synthesized for the operations by application of the methods discussed in Chapter 3 are shown in Figure 58. Operation 2 has three direct predecessors, the PI operation and operations 3 ' and 5 ' from the preceding cycle. Operation 2 dominates operation PI and the output of 5 ' that reaches operation 2 but it does not dominate operation 3 '. A wait operation is added whose decision outputs come from 5 ' and whose signal input has operation 3 ' as a predecessor. The wait operation output corresponding to the decision outcome that reaches operation 2 is dominated by operation 2 and depends on the completion of operations 5 ' and 3 '. This signal and the one from the PI node form a cut set for operation 2 that includes all predecessors of operation 2. Since this is the only cut set for operation 2, these signals must be merged together as shown in Figure 58 to form the initiation network for operation 2. Next the initiation network for the signal input to the wait operation is determined and it consists of the completion signal from operation 2. The remaining initiation networks are synthesized in a similar manner and are shown in Figure 58. Figure 59 shows the combination of all of these initiation networks with branch operations added where the output from an operation is required by more than one following operation.

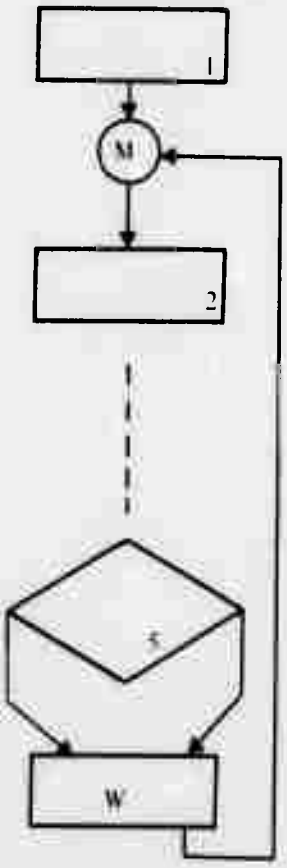
If the time required for operations 4 and 5 is longer than that required for operations 2 and 3 the system will operate satisfactorily. However, if operations 4 and 5 require less time than operations 2 and 3, there may be several signals applied to input C of the wait operation before a signal is applied to input A.

The problem occurs because input C of the wait operation may receive a signal from the second cycle of the process before input A receives a signal from the first cycle. The error would be eliminated if the wait operation could store each input signal to B and C until the corresponding signal was received by input A. However, since the number of cycles to be executed and the relative speeds of the operations are not known, the number of input signals that the wait operation would be required to store may be arbitrarily large.

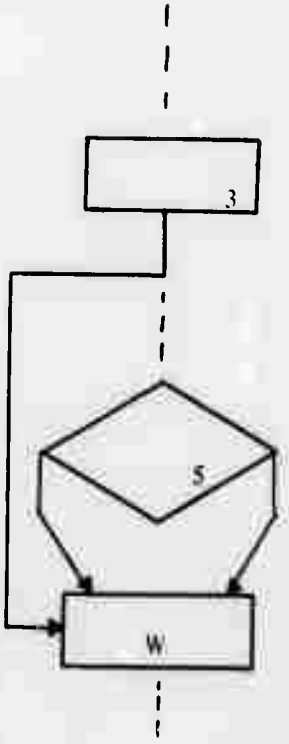
The error can also be eliminated by making operation 5 depend on the completion signal from the wait operation that corresponds to input C as shown in Figure 60. In this case only one signal can be received by the wait operation at input C before a signal is received at input A, and the wait operation completion signal generated.

Other types of errors can also occur as Figure 61 shows. If the execution of operation 2 is slow enough, the right input to the rendezvous at the input to operation 3 will receive two input signals before the left input of the rendezvous receives any signals. In this case the input signal to the rendezvous for a given cycle can be generated before the completion signal has been generated for the preceding cycle and a hazard exists. There are several changes that can be made to eliminate the error, one of which is shown in Figure 62. The error is eliminated by insuring that the rendezvous input signal for any cycle depends on the rendezvous completion signal from the preceding cycle.

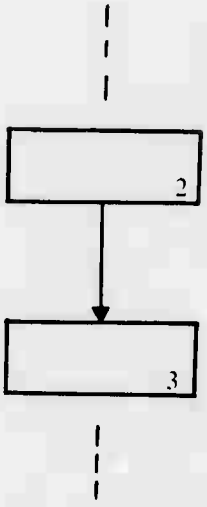
In the following section a proof is presented to show that if each signal is self-dependent (the initiation of the signal on any cycle depends on the completion of the signal from all preceding cycles) the process that is synthesized will be free of implementation errors.



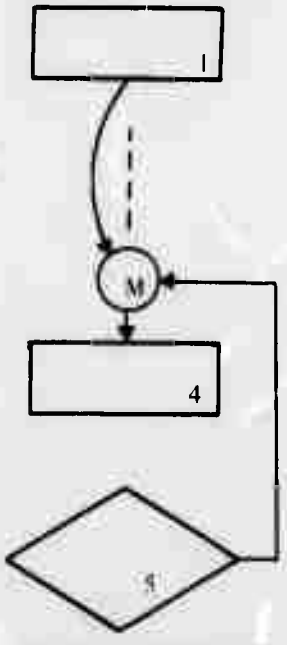
INITIATION NETWORK
FOR OPERATION 2



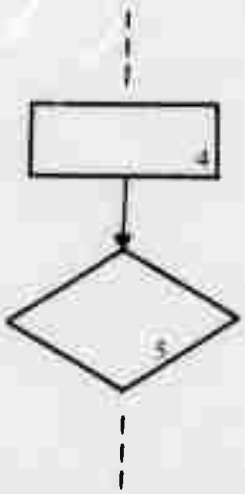
INITIATION NETWORK
FOR THE WAIT OPERATION



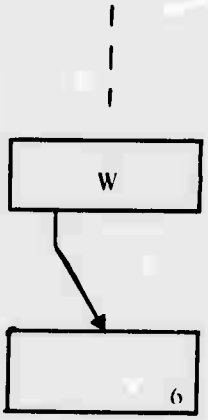
INITIATION NETWORK
FOR OPERATION 3



INITIATION NETWORK
FOR OPERATION 4



INITIATION NETWORK
FOR OPERATION 5



INITIATION NETWORK
FOR OPERATION 6

FIGURE 58. THE INITIATION NETWORKS FOR THE EXAMPLE OF FIGURE 57

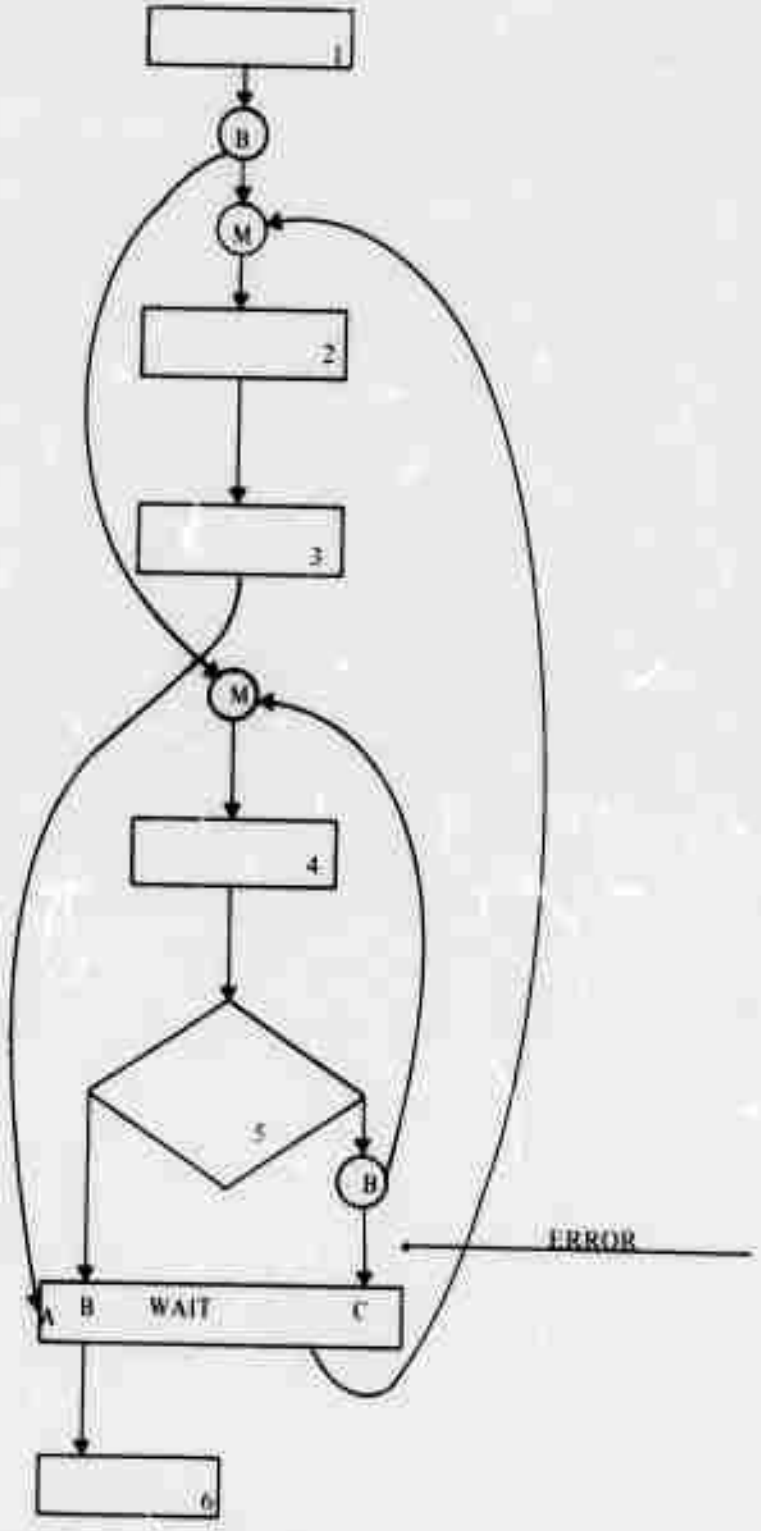
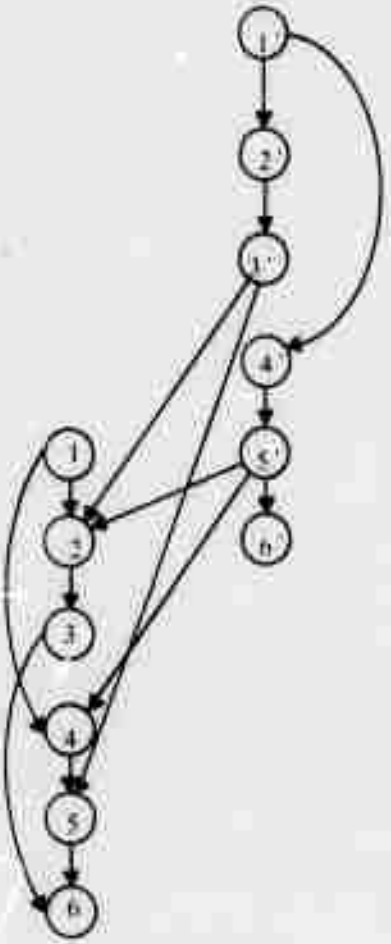
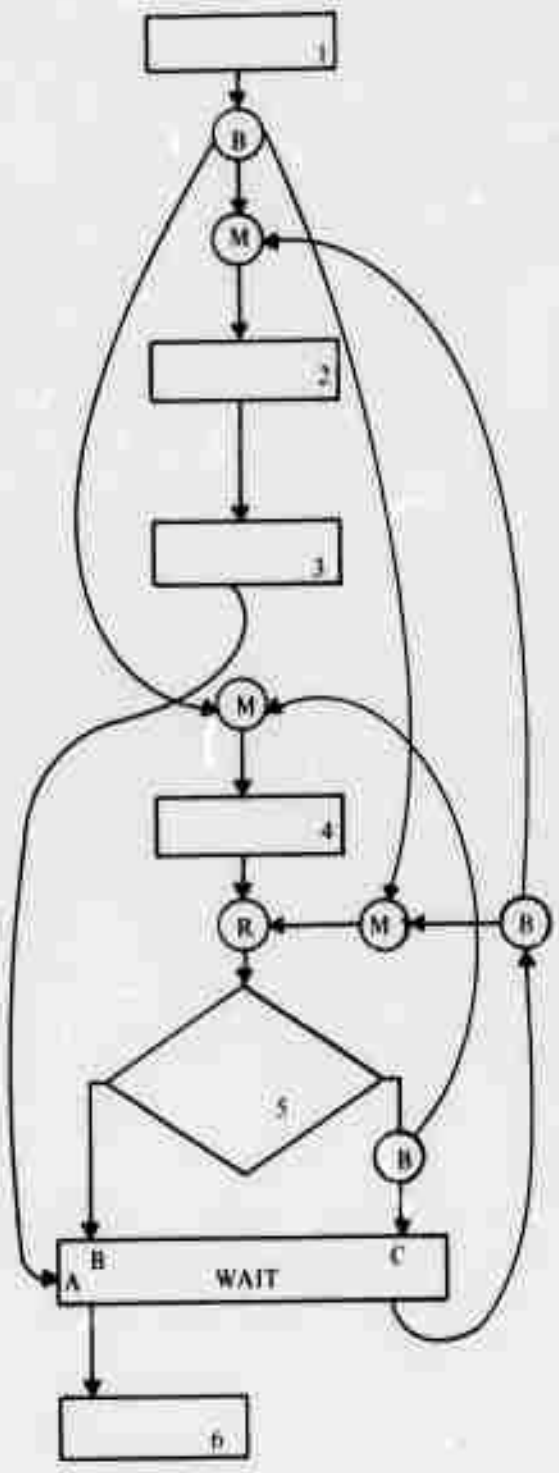


FIGURE 59. COMPLETE CONCURRENT FLOW CHART FOR THE EXAMPLE OF FIGURE 57



PRECEDENCE GRAPH



FLOW CHART

FIGURE 60. THE PROCESS OF FIGURE 59 WITH OPERATION 5 DEPENDING ON THE COMPLETION OF THE WAIT OPERATION

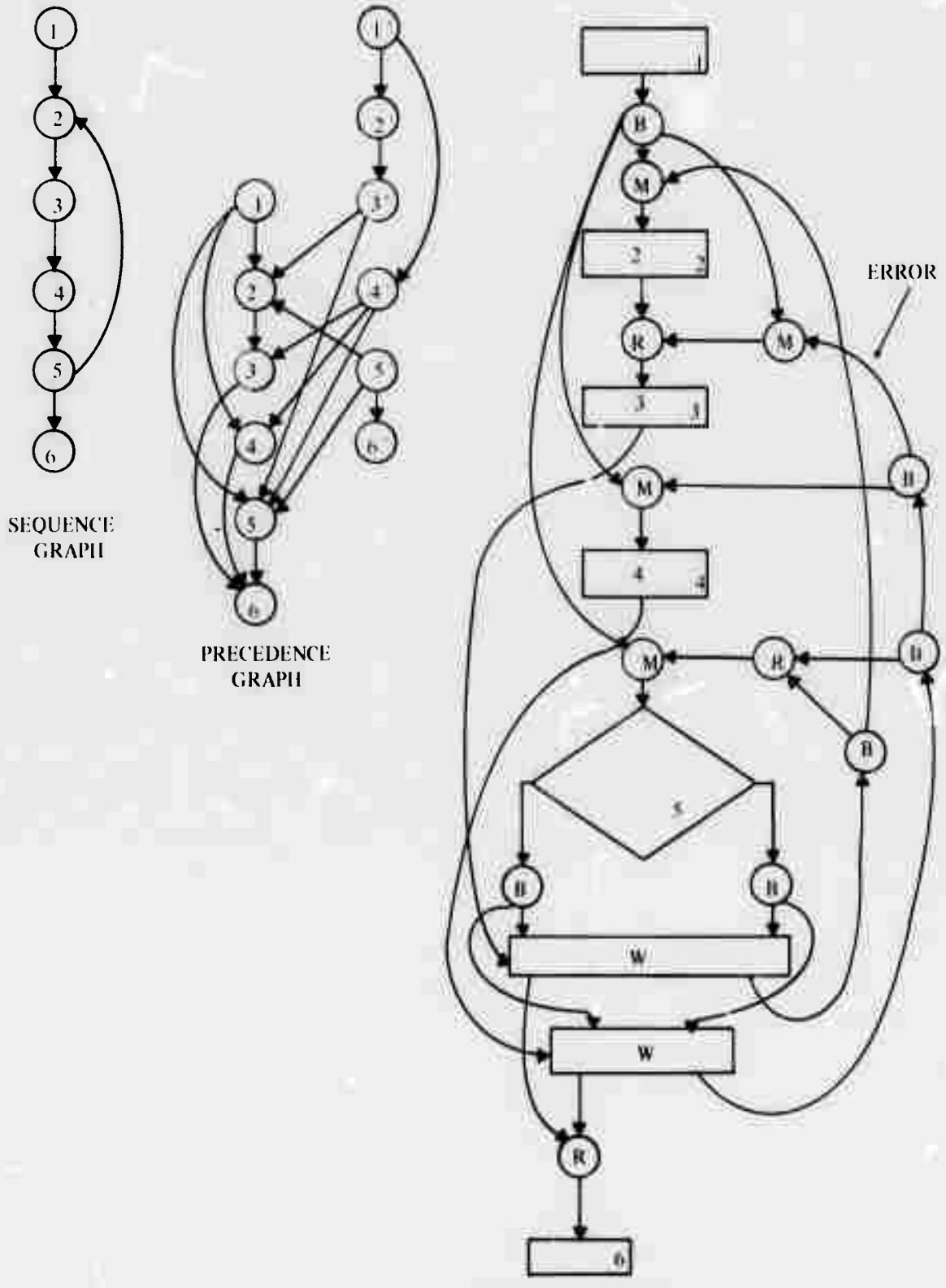


FIGURE 61. A CONCURRENT PROCESS WITH A HAZARD

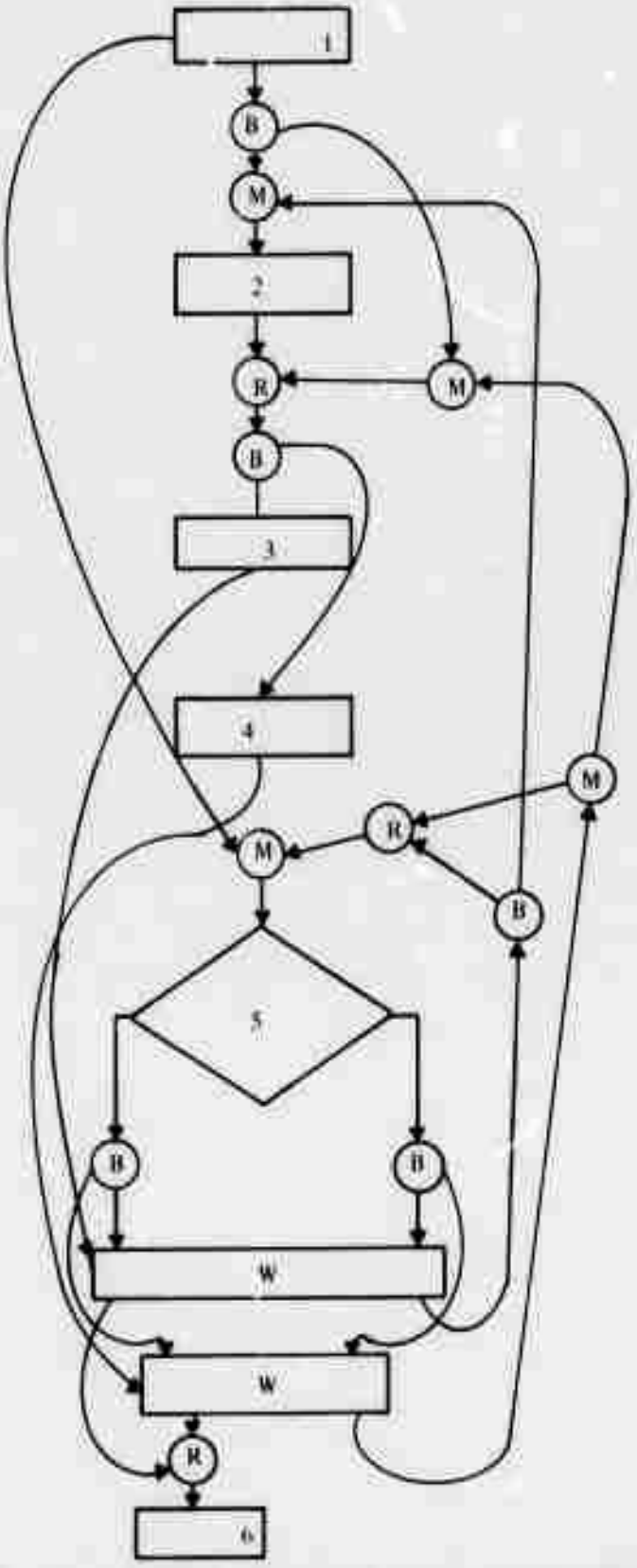


FIGURE 62. THE PROCESS OF FIGURE 61 WITH THE ERROR CORRECTED

4.7.1 Self-Dependent Signals

In this section we will show that the requirement for all signals to be self-dependent (the initiation of each signal depends on the completion of that signal from all preceding cycles) in conjunction with the previous requirements that an operation must dominate all of its direct predecessors and that cut sets are used to determine combinations of signals that are merged together, is necessary and sufficient to guarantee a concurrent process that is free of implementation errors. The specific types of implementation errors considered are the incomplete rendezvous and hazard as discussed in Chapter 3.

Theorem 6: The requirement that all signals of a process be self-dependent is both a necessary condition, and in conjunction with the requirements on synthesis of M–R networks, a sufficient condition for a synthesized concurrent process to be free of implementation errors.

Proof: To show the necessary part consider any operation, X, that is not self-dependent. X cannot depend on the completion signal of operation X from every preceding cycle or it would be self-dependent. Then operation X may receive an initiation signal before it generates a completion signal for some preceding cycle and this represents a hazard. Thus, if any operation in a process is not self-dependent, a hazard exists in the process.

To show that the requirement is sufficient, consider any process in which no operation can receive an input signal for a particular cycle until its execution is completed for all preceding cycles. That is, on every cycle except the first, an operation cannot be initiated until it has been completed on all preceding cycles, or if it is not executed on a preceding cycle, until everything that its execution in the preceding cycles would depend on, has been completed. Figure 63 shows a typical operation, I, and the control network used to generate the initiation signal for I. The inputs to the merge operations may come from operations in the same cycle of the process as I or from operations in preceding cycles but all of the operations must be dominated by I.

Each input to one of the merge operations depends on the completion of the merge operation on all preceding executions of that merge operation. Since the only place the completion of the merge operation can be tested is the output of operation I, operation I must be completed or else not executed on every preceding cycle before any input signal for the merge operation can be generated for the current cycle.

Consider the first execution of operation I. Each merge operation in the M–R network will receive exactly one input corresponding to that execution of I since exactly one member of each of I's cut sets must be executed for each execution of operation I. The merge operations cannot receive any additional signals until operation I is completed and therefore are free of hazard errors for the first execution of operation I. The rendezvous will receive one initiation signal on each of its inputs and cannot receive any further

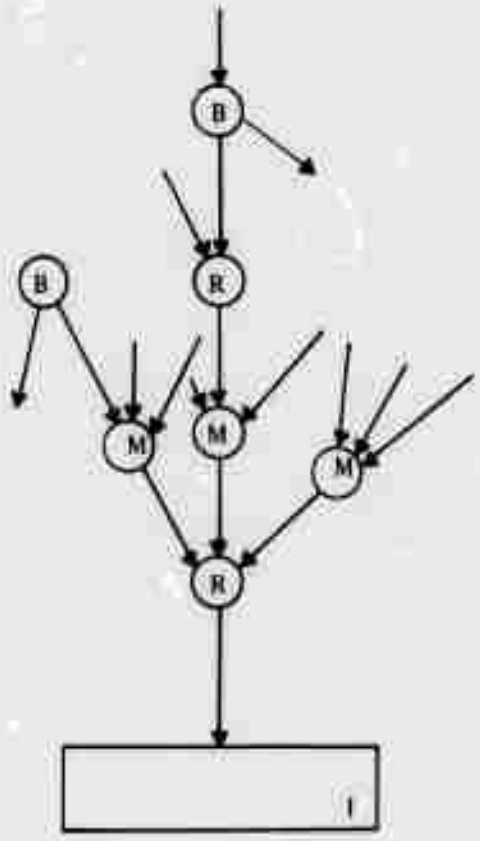


FIGURE 63. A TYPICAL OPERATION AND ITS M R NETWORK

initiation signals until its execution is complete and the execution of operation I is complete. Therefore the rendezvous is free of incomplete rendezvous and hazard errors. Operation I is also free of hazards since it cannot receive a second initiation signal until its execution is completed. Thus operation I and its M–R network must be free of implementation errors the first time that they are executed.

Next, consider the execution of operation I on cycle M when all preceding executions have been error-free. Before any merge operation can receive an input signal corresponding to the execution of I on cycle M, the execution of I must be completed on all preceding cycles, and since these preceding executions are error-free no merge operations or rendezvous inputs can be active. As before, each merge operation must receive one input signal and can receive no additional signals until operation I is completed so the M–R network will be free of hazard and incomplete rendezvous errors. Thus there will be no hazard or incomplete rendezvous errors for operation I and its M–R network on the first execution of I or any execution of I that follows an error-free execution of I. The same proof can be extended slightly to account for the additional inputs and used to show that there are no hazard or incomplete rendezvous errors in any wait operation or its M–R network.

Next, consider a rendezvous operation that is used to combine two or more signals from direct predecessors of operation I that have the same dominance relation. If either of the direct predecessors is executed the other one must be executed also and neither can be executed twice before the completion signal from the rendezvous is generated since both of the direct predecessors must depend on the completion of operation I which depends on the rendezvous completion. Thus there can be no incomplete rendezvous or hazard errors in the rendezvous operation.

The only other operations in the system are branch operations but there can be no hazard in a branch operation unless there is also a hazard in the operation from which the branch operation receives its input. Thus there can be no hazard or incomplete rendezvous errors in the synthesized process if all operations are self-dependent.

4.7.2 Synthesis of Processes with Self-Dependent Signals

The preceding section has shown that all signals must be self-dependent if the synthesized process is to be error-free. Next, methods must be established to insure that all signals in any synthesized process are self-dependent. The following sections present several methods which were investigated to assure that the control network synthesized from a precedence and connectivity relation would contain only self-dependent signals.

4.7.2.1 Reciprocal Precedence Relations

By a reciprocal precedence relation it is meant that if an operation, J, is a predecessor of an operation, I, then I must be a predecessor of J whenever J follows I. Figure 64 shows examples of portions of several precedence graphs,

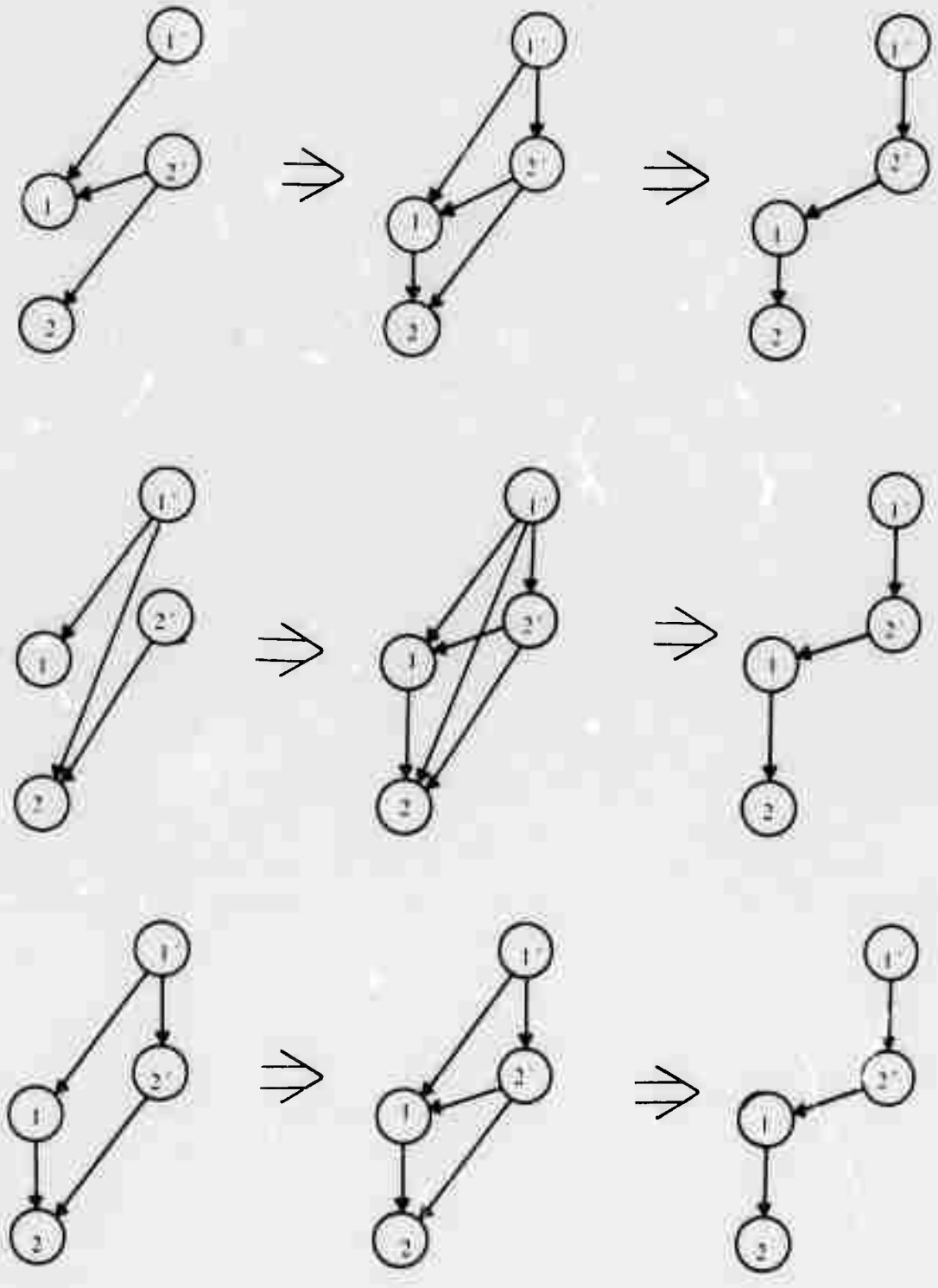


FIGURE 64. FORMATION OF THE RECIPROCAL PRECEDENCE RELATION

the added members of the precedence relation used to make the precedence relation reciprocal, and the final precedence relation with all implied members deleted. The requirement that the precedence relation be reciprocal is sufficient to insure that each input signal to an operation depends on the completion of that operation for all preceding initiations. Unfortunately, the reciprocal precedence relation also causes the execution of all operations within a loop to be completely sequential as is shown below.

Every operation in a loop must have the loop exit decision from the previous execution of the loop as a predecessor. Then if the precedence relation is to be reciprocal the loop exit decision must have every operation in the loop as a predecessor. Then each operation in one execution of the loop has every operation in a preceding execution of the loop as a predecessor. Then each operation has every operation in its cycle of the loop that precedes it as a predecessor also, and the operations must be executed sequentially.

4.7.2.2 Reciprocal Basic Precedence Relations

The requirement that all precedence relations be symmetric, as discussed in the last section, is sufficient to provide an error-free network but also eliminates concurrent operation within loops. Less restrictive requirements are needed that will still assure an error-free network. One possibility, which will be investigated in this section, is to require only the basic precedence relation to be reciprocal. That is, if an operation, J, is a direct predecessor of an operation, I, then there must be a member of the precedence relation (not necessarily the basic precedence relation) between I and J whenever I follows J. Since every operation that provides a signal for the M–R network of an operation, I, is a direct predecessor of I, this will require all preceding executions of I to be completed before any of I's direct predecessors can be executed. Thus, all signals will be self-dependent and the synthesized process will be free of implementation errors. The required members of the precedence relation can always be added if there is a path from I to J that contains fewer than two feedback arcs. If there is no path containing fewer than two feedback arcs then the precedence relation as developed in Section 4.3 will require I to be executed before J.

Following is a summary of the steps used to synthesize processes with type C concurrency.

1. Determine the dominance relation as described in Section 4.2.
2. Determine the precedence relation for two copies of the system as described in Section 4.3.
3. Select an operation, I, other than the PI operation, each of whose direct predecessors has either had its M–R network synthesized, is in the preceding copy of the process, or is the PI node.
4. If any direct predecessor, K, of I is not subordinate to I, find the closest decision to K that can be reached by K, that reaches I, and that has an output that is dominated by I. Add a wait operation that receives its decision inputs from the decision and make K a predecessor of the wait operation and the wait operation a predecessor of operation I. Update the direct precedence relation as described in Section 3.3.4, including any relations required to make the basic precedence relation reciprocal and return to step 3.
5. If any direct predecessors of I have the same dominance relation, connect their outputs to the inputs of a rendezvous operation and use the rendezvous output in place of the operation outputs.
6. Determine the cut sets for I and use a cut set table to choose a group of cut sets that cover all of the predecessors of I and synthesize the corresponding M–R network.
7. If operation I is not the PT operation, return to step 3.

4.7.2.3 Example

As an example of the application of this method consider the sequential flow chart shown in Figure 65. Operations 2, 3, and 4 refer to storage locations A to D and operation 5 decrements a count and tests it to determine whether the loop should be repeated or not. Figure 66 shows the basic precedence relation for two copies of the process. All of the members of the precedence relation are determined as described in Section 4.2 except the one from operation 2 to operation 5 which is included to make the basic precedence relation reciprocal. Since operation 2 does not dominate operation 3' a wait operation must be added and it will receive its decision inputs from operation 5. The wait operation will have two outcomes and the updated precedence relation is shown in Figure 66 where the node representing the wait operation is drawn to emphasize the two outcomes. Another wait operation must be added since operation 3 does not dominate operation 4' and the complete precedence relation is shown in Figure 66. The flow chart of the synthesized concurrent process is shown in Figure 67 and it shows that operation 2 of a given cycle may be initiated before operation 4 of the preceding cycle has been completed. The flow chart can be simplified slightly and some of the merge, branch, and rendezvous operations eliminated by observing that some signals are required as inputs to several operations. For example, the left output from W2 could be merged with the output signal from operation 1 and then branched to provide signals for the rendezvous associated with operations 3 and 5, thereby saving one merge and one branch operation.

The concurrent process in Figure 67 includes several unnecessary restrictions. For example, operation 5 of a given cycle must follow operation 2 of the same cycle due to the member of the precedence relation that was added to make the precedence relation reciprocal. It is obviously not necessary that operation 5 follow operation 2 of the same cycle. In order to allow the synthesis of a process in which operations 2 and 5 can be executed concurrently a dummy operation, X, can be added directly before operation 2. X is made a predecessor of operation 2 and every operation that was originally a predecessor of operation 2 is made a predecessor of operation X. The basic precedence relation is shown in Figure 68 with the added member required to make it reciprocal. Note that with the addition of operation X, operations 2 and 5 can be executed concurrently.

Another place where the members added to the precedence relation to make it reciprocal reduce the possibilities of concurrent operation is shown in Figure 66. The members of the precedence relation from W1' and W2' to operation 5 prevent operation 5 from being executed until operations 3 and 4 of the preceding cycle are completed. Actually, there is nothing in the original process that restricts operation 5 from being executed any number of times before operations 2, 3, and 4 are completed the first time. In order to allow operation 5 to be executed before operation 3 of the preceding cycle is completed a pair of wait operations can be used in place of W1 as is shown in Figure 69. Two wait operations, W1A and W1B, are added to the precedence relation as shown in Figure 69 to generate a signal that depends on the completion of operation 3 from the preceding cycle and that is subordinate to operation 2. The direct precedence relation can then be made reciprocal without requiring operation 5 to depend on the completion of operation 3 from the preceding cycle. By replacing W2 with a pair of wait operations in the same manner the dependence of operation 5 on operation 4 of the preceding cycle can also be eliminated. The execution of operation 5 on cycle N will then depend on the completion of operations 3 and 4 on cycle N–2.

By using enough wait operations operation 5 may be allowed to proceed any number of cycles ahead of operations 3 and 4 but the number of wait operations and complexity of the synthesized process increase significantly. The wait operations allow the execution of operation 5 to proceed ahead of the other operations by storing the outcome of the decision. This is an inefficient method of storing this information and a system of

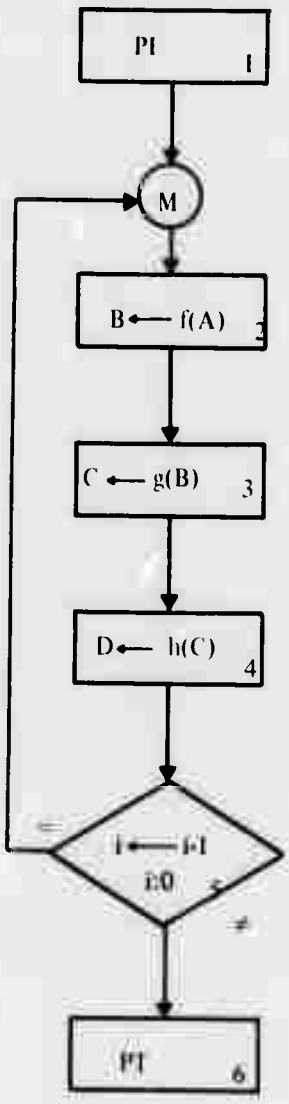


FIGURE 65. A SEQUENTIAL FLOW CHART WITH ONE LOOP

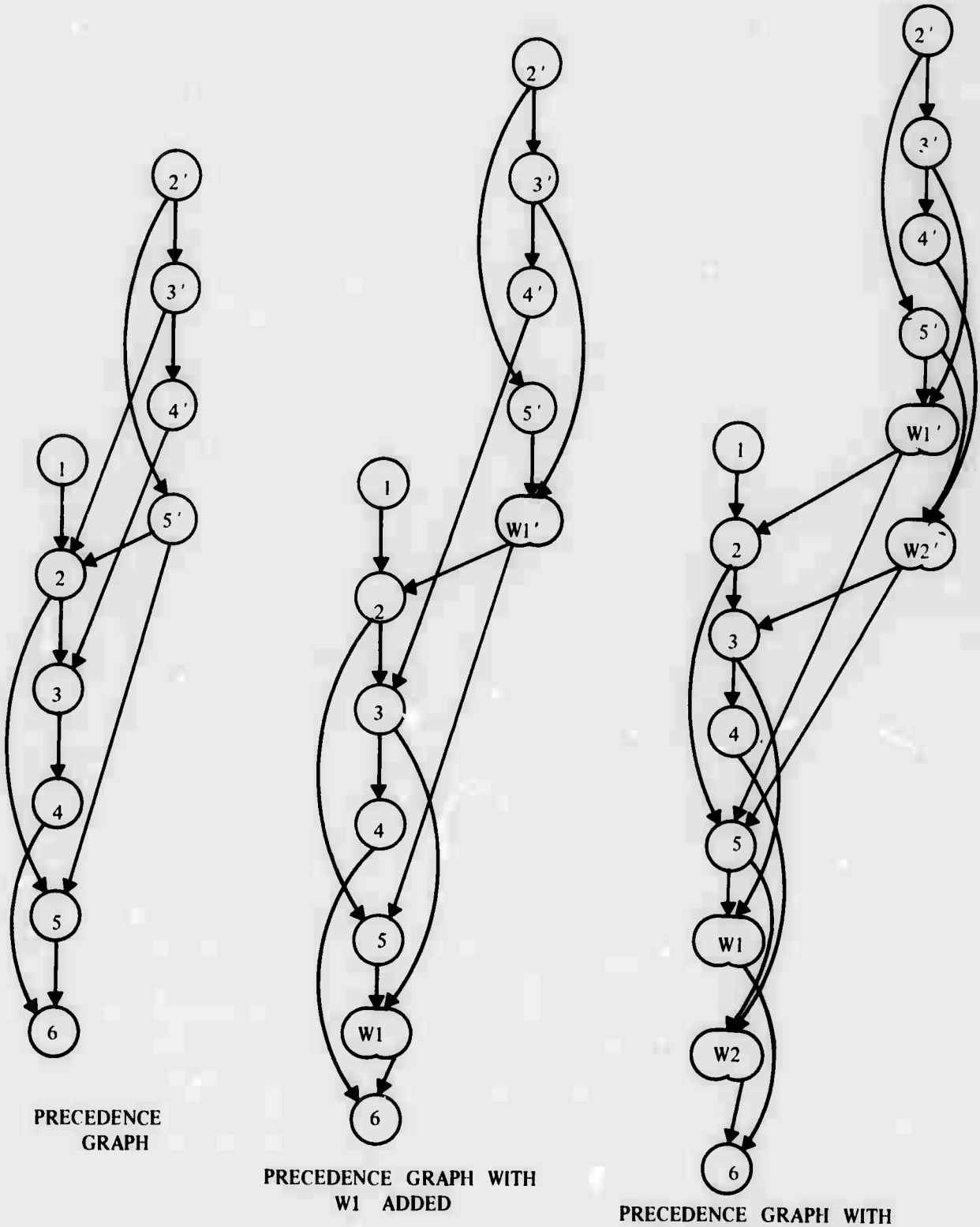


FIGURE 66. PRECEDENCE GRAPHS FOR THE FLOW CHART OF FIGURE 65

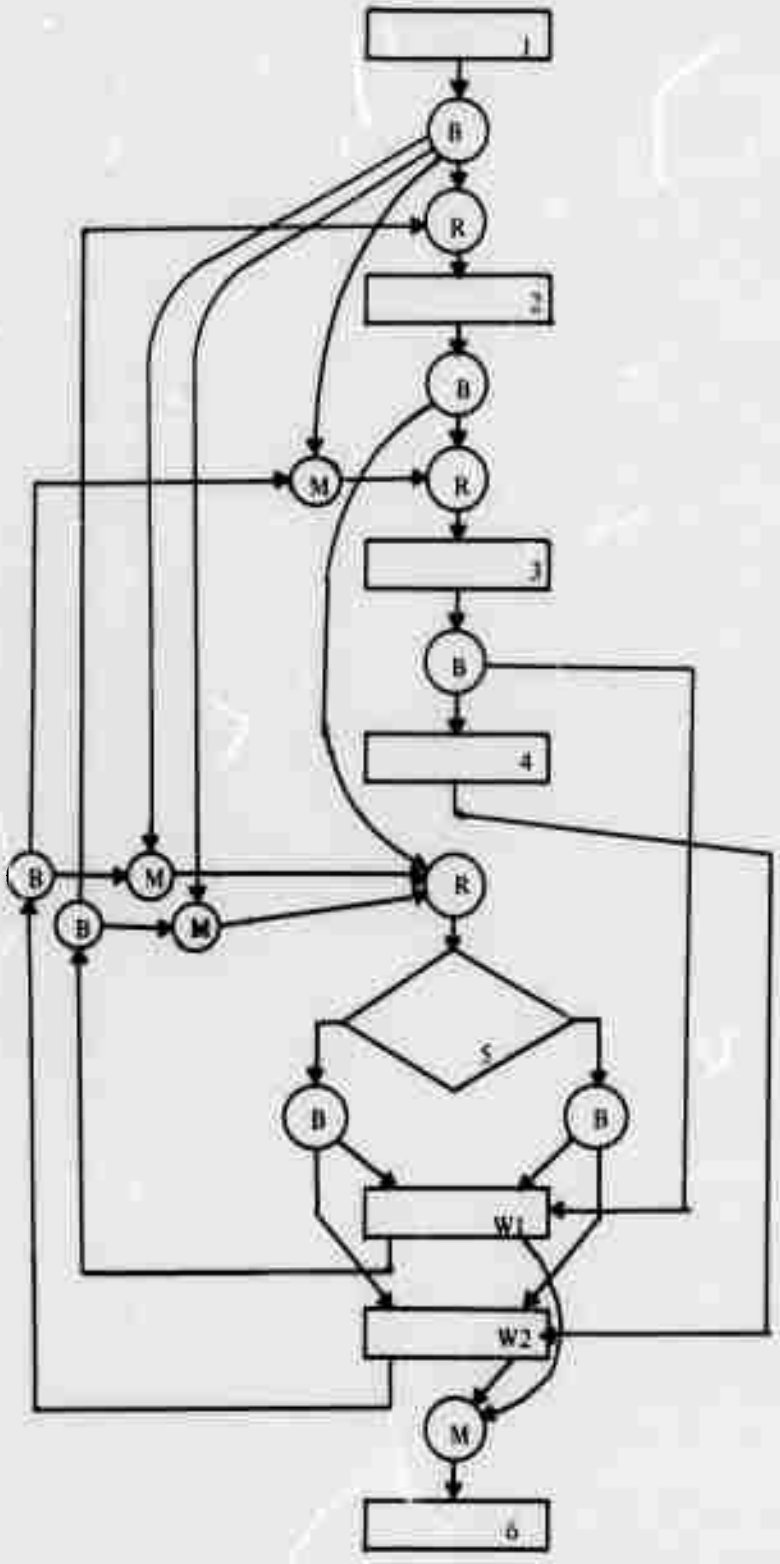


FIGURE 67. A CONCURRENT FLOW CHART FOR THE EXAMPLE OF FIGURE 65

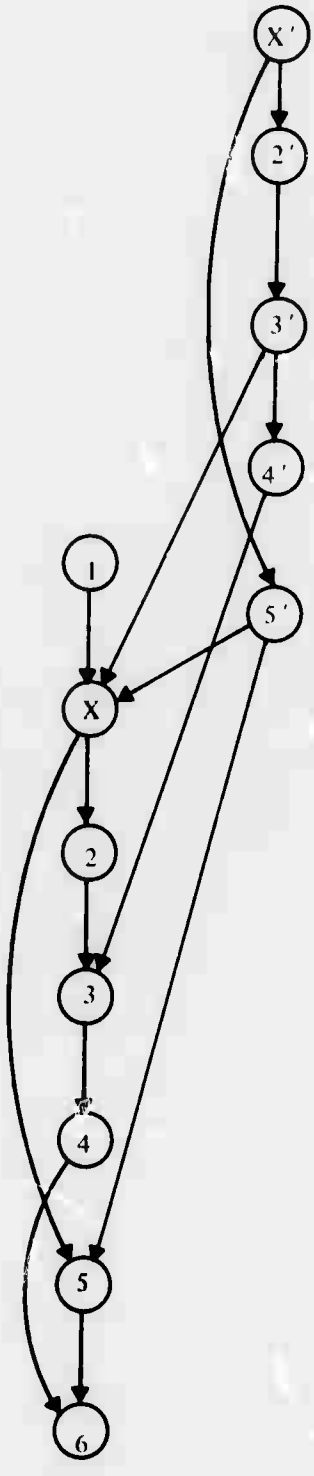
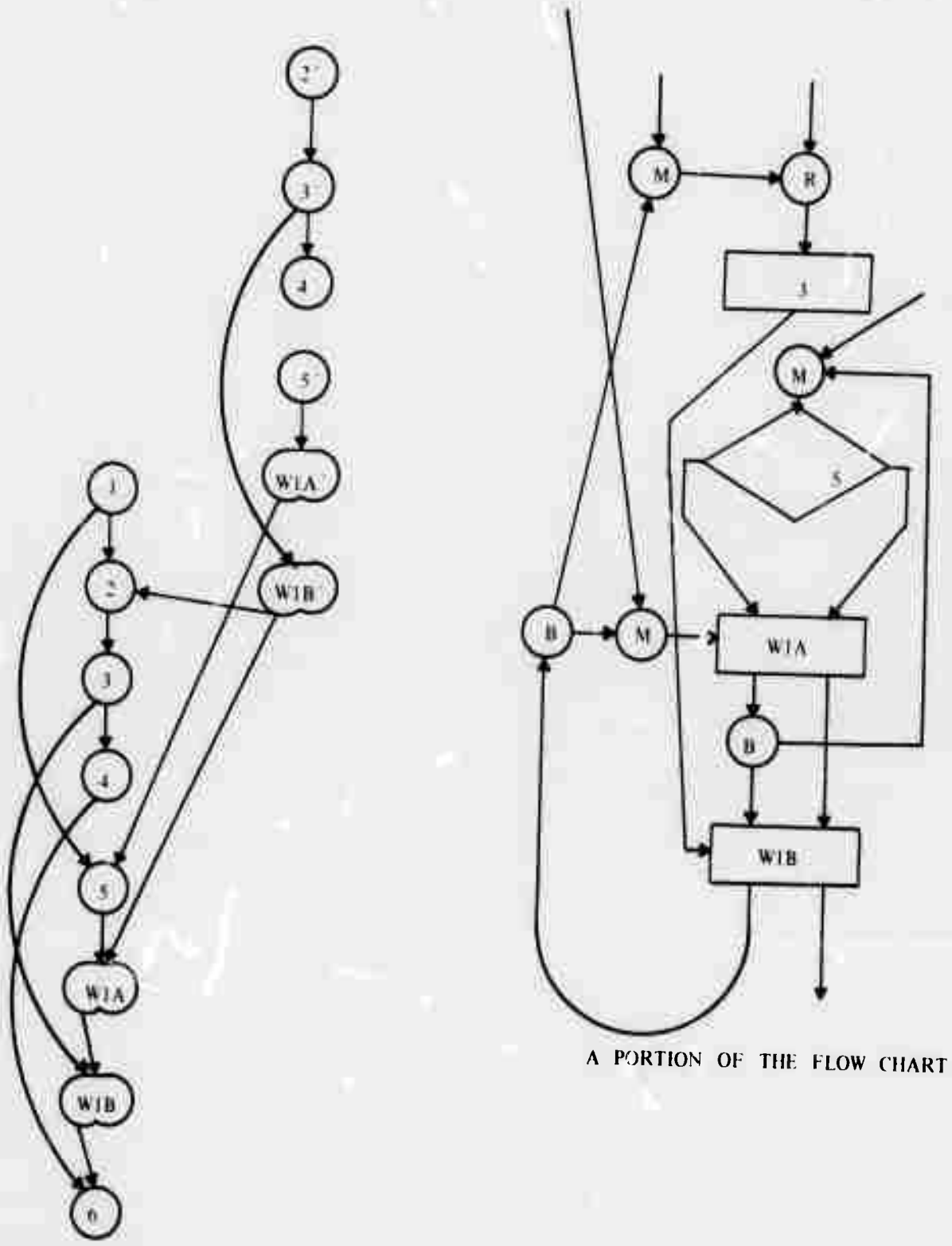


FIGURE 68. PRECEDENCE GRAPH FOR THE EXAMPLE OF FIGURE 65 WITH DUMMY OPERATION X ADDED



PRECEDENCE GRAPH

A PORTION OF THE FLOW CHART

FIGURE 69. AN EXAMPLE OF THE USE OF A PAIR OF WAIT OPERATIONS TO ALLOW ADDITIONAL CONCURRENCY

counters or data queues as discussed by Karp and Miller and by Reiter[20, 21] would be more efficient. However, they have discussed only methods for analysis of their systems and not for synthesis.

4.8 SUMMARY

Three methods have been presented for synthesizing concurrent processes from sequential processes containing loops. The first method, which consists of treating the execution of each cycle of the process independently, is the easiest to apply but does not allow any cycle-to-cycle concurrency. The second method allows some cycle-to-cycle concurrency while the third method, which consists of forming a reciprocal precedence relation is the most complex but may produce a faster system since additional cycle-to-cycle concurrency is possible. Which of the three methods is best for a given process will depend on the particular characteristics of the process including the execution times of the individual operations.

5. CONCLUSION

Methods have been developed which can be used to analyze a sequential process and to then synthesize an error-free process that performs the same calculations but in less time by performing some operations concurrently. Precedence relations, as developed by Bernstein[26] and Fisher[9] are reviewed in Sections 2.4 and 4.3 and are used to describe any required order of execution of the operations comprising the process. The dominance relation of Prosser[30] is extended to include processes with loops and is used to represent the relation between the execution of operations. Wait operations are added to the original process until every direct predecessor of an operation is subordinate to the operation and an M-R network is then used to allow initiation of the operation as soon as all of the operation's predecessors are completed. The precedence relation is used to prevent sequencing errors by insuring that any required order of execution is observed while the dominance relation and directed cut sets are used to prevent implementation errors. The M-R networks synthesized are analogous, although not isomorphic, to or-and switching networks and some of the techniques developed for synthesis of multilevel and multioutput or-and switching networks might be applied to simplify the M-R networks. Several methods were developed, varying in complexity and effectiveness, for treating processes with loops.

The synthesis methods described are by no means exhaustive and other possibilities exist which may allow simpler processes to be synthesized by reducing the possibilities for concurrent operation. For example, in some cases a number of wait operations may be required which significantly complicate a concurrent process while allowing a very small or perhaps no reduction in the execution time of the process. If an operation, I, has an operation, J, as a direct predecessor but I does not dominate J the use of wait operations can be avoided by modifying the synthesis procedure slightly. Instead of adding a wait operation, operation J can be made a predecessor of the decision from which the wait operation would have received its decision inputs. Operation I will have the decision as a predecessor and the decision output will be dominated by I. This will eliminate the requirement for the wait operation and will prevent operation J from being executed concurrently with the decision or with any operation that has the decision as a predecessor. Thus the synthesized process will be simpler and will allow less concurrent execution of the operations.

The only other effort to synthesize concurrent processes known to the author is that of Bingham *et. al.*[51,52] Their method consists of maintaining the precedence matrix and several binary vectors in memory and using these to control the sequencing of the operations. Each time an operation is completed its corresponding bit is set in a vector specifying all completed operations. The precedence graph is then checked to find all operations whose predecessors have been completed and these are added to a vector representing operations ready for execution. Whenever a decision is executed all operations that cannot be reached by the selected outcome are marked as having been completed so that their successors may be initiated. Although it can probably be modified to include processes with loops, the method as presented does not include processes with loops. Some method would be required to reinitialize or else to provide additional copies of the status vectors when a loop is repeated. The use of status vectors and the precedence matrix during the process execution does not seem practical in a hardware process such as might be constructed from macromodules but it could be implemented in a programmed system.

Additional work is required to allow effective use of concurrent operations. A difficult but necessary requirement is the ability to determine where concurrent execution of operations is worth the added complexity and where it is not. This obviously requires estimates of the execution time of the individual operations and also requires knowledge

of the number of times that each operation is to be executed if the times to execute sections of the process are to be estimated. This problem has been studied by Estrin and Martin[53] and by Kral[54] but is extremely difficult to analyze since branching probabilities may be data dependent. Another problem is the assignment of operations to the hardware or processors that will be used to execute them. As the example in Figure 10 shows, the use of additional storage locations or reassignment of storage locations may affect the degree of concurrent execution possible.

6. BIBLIOGRAPHY

1. Winograd, S., "On the Time Required to Perform Addition", *Journal of the Association for Computing Machinery*, Vol. 12, No. 2, (277–285), April 1965.

2. Winograd, S., "On the Time Required to Perform Multiplication", *Journal of the Association for Computing Machinery*, Vol. 14, No. 4, (793–802), October 1967.

3. Winograd, S., "How Fast Can Computers Add", *Scientific American*, Vol. 219, No. 4, (93–100), October 1968.

4. Conway, M.E., "A Multiprocessor System Design", *Proceedings of the Fall Joint Computer Conference*, (139–146), 1963.

5. Anderson, J.P., "Program Structures for Parallel Processing", *Communications of the Association for Computing Machinery*, Vol. 8, No. 12, (786–788), December 1965.

6. Dennis, J.B., and Van Horn, E.C., "Programming Semantics for Multiprogrammed Computations", *Communications of the Association for Computing Machinery*, Vol. 9, No. 3, (143–155), March 1966.

7. Gosden, J.A., "Explicit Parallel Processing Description and Control in Programs for Multi and Uni-Processor Computers", *Proceedings of the Fall Joint Computer Conference*, (651–660), 1966.

8. Opler, A., "Procedure-Oriented Language Statements to Facilitate Parallel Processing", *Communications of the Association for Computing Machinery*, Vol. 8, No. 5, (306–307), May 1965.

9. Fisher, D.A., *Program Analysis for Multiprocessing*, TR 67–2, Burroughs Corporation, Great Valley, Pennsylvania, May 1967.

10. Clark, W.A., "Macromodular Computer Systems", *Spring Joint Computer Conference*, (335–336), 1967.

11. Ornstein, S.M., Stucki, M.J., and Clark, W.A., "A Functional Description of Macromodules", *Spring Joint Computer Conference*, (337–355), 1967.

12. Stucki, M.J., Ornstein, S.M., and Clark, W.A., "Logical Design of Macromodules", *Spring Joint Computer Conference*, (357–364), 1967.

13. Blum, A., Chaney, T.J., and Olsen, R.E., "Engineering Design of Macromodules", *Spring Joint Computer Conference*, (365–370), 1967.

14 Wann, D.F., *Error Analysis in Parallel Processing*, Computer Systems Laboratory, Washington University, St. Louis, Missouri, Technical Memorandum No. 15, December 1966.

15. Turn, R., *Assignment of Inventory of a Variable Structure Computer*, Department of Engineering, University of California, Los Angeles, Technical Report No. 63–5, January 1963.

16. Estrin, G., and Turn, R., "Automatic Assignment of Computations in a Variable Structure Computer System", *Institute of Electrical and Electronic Engineers Transactions on Electronic Computers*, Vol. EC–12, No. 6, (755–773), December 1963.

17. Martin, D.F., *The Automatic Assignment and Sequencing of Computations on Parallel Processor Systems*, Department of Engineering, University of California, Los Angeles, Report 66–4, January 1966.

18. Martin, D.F., and Estrin, G., "Experiments on Models of Computations and Systems", *Institute of Electrical and Electronic Engineers Transactions on Electronic Computers*, Vol. EC–16, No. 1, (59–69), February 1967.

19. Schwartz, E.S., "A Heuristic Procedure for Parallel Sequencing with Choice of Machines", *Management Science*, Vol. 10, No. 4, July 1964.

20. Karp, R.M., and Miller, R.E., "Properties of a Model for Parallel Computations: Determinacy, Termination, Queing", *SIAM Journal of Applied Math.*, Vol. 14, No. 6, (1390–1411), November 1966'

21. Reiter, R., "Scheduling Parallel Computations", *Journal of the Association for Computing Machinery*, Vol. 15, No. 4, (590–599), October 1968.

22. Wann, D.F., Ellis, R.A., Stucki, M.J., and Keller, R.M., "Problems Encountered with Control Networks in Highly Restructurable Digital Systems", *Digest of the Institute of Electrical and Electronic Engineers Computer Conference*, (29–32), September 6–8, 1967.

23. Keller, R.M., and Wann, D.F., *Analysis of Implementation Errors in Digital Computing Systems*, Computer Systems Laboratory, Washington University, St. Louis, Technical Report No. 6, March 1967.

24. Van Horn, E.C., *Computer Design for Asynchronously Reproducible Multiprocessing*, AD 650 407, September 1966.

25. Van Horn, E.C., "Three Criteria for Designing Computing Systems to Facilitate Debugging", *Communications of the Association for Computing Machinery*, Vol. 11, No. 5, (360–365), May 1968.

26. Bernstein, A.J., "Analysis of Programs for Parallel Processing", *Institute of Electrical and Electronic Engineers Transactions on Electronic Computers*, Vol. EC–15, No. 5, (757–763), October 1966.

27. Thorelli, L., "An Algorithm for Computing all Paths in a Graph", *BIT* Vol. 6, No. 4, (347–349), 1966.

28. Mann, W.C., *A Data Structure for Directed Graphs in Man-Machine Processing*, AD 636 251, June 20, 1966.

29. Jenson, P.A., *A Graph Decomposition Technique for Structuring Data*, AD 658 756, September 1967.

30. Prosser, R.T., "Applications of Boolean Matrices to the Analysis of Flow Diagrams", *Proceedings of the Eastern Joint Computer Conference*, (133–138), 1959.

31. Warshall, S., "A Theorem on Boolean Matrices", *Journal of the Association for Computing Machinery*, Vol. 9, No. 1, (11–12), January 1962.

32. Fisher, A.C., Liebman, J.S., and Nemhauser, G.L., "Computer Construction of Project Networks", *Communications of the Association for Computing Machinery*, Vol. 11, No. 7, (493–497), July 1968.

33. Chen, Y.C., and Wing, O., "Some Properties of Cycle-Free Directed Graphs and Identification of the Longest Path", *Journal of the Franklin Institute*, Vol. 281, No. 4, (293–301), April 1966.

34. Ramamoorthy, C.V., "Analysis of Graphs by Connectivity Considerations", *Journal of the Association for Computing Machinery*, Vol. 13, No. 2, (211–222), April 1966.

35. Ramamoorthy, C.V., "A Structural Theory of Machine Diagnosis", *Spring Joint Computer Conference*, (743–756), 1967.

36. Bingham, H., Fisher, D., and Semon, W., *Detection of Implicit Computational Parallelism for Input Output Sets*, AD 645 120, December 1966.

37. Bingham, H., Fisher, D., and Semon, W., *Detection of Essential Ordering Implicit in Compiler Language Programs*, AD 650 845, February 1967.

38. Nievergelt, J., "On the Automatic Simplification of Computer Programs", *Communications of the Association for Computing Machinery*, Vol. 8, No. 6, (366–370), June 1965.

39. Dijkstra, E.W., "The Structure of the 'THE' - Multiprogramming System", *Communications of the Association for Computing Machinery*, Vol. II, No. 5, (341–346), May 1968.

40. Dorn, W.S., "Generalization of Horner's Rule for Polynomial Evaluation", *IBM Journal of Research and Development*, Vol. 6, No. 2, (239–245), April 1962.

41. Nievergelt, J., "Parallel Methods for Integrating Ordinary Differential Equations", *Communications of the Association for Computing Machinery*, Vol. 7, No. 12, (731–733), December 1964.

42. Shedler, G.S., "Parallel Numerical Methods for the Solution of Equations", *Communications of the Association for Computing Machinery*, Vol. 10, No. 5, (286–291), May 1967.

43. Karp, R.M., Miller, R.E., and Winograd, S., "The Organization of Computations for Uniform Recurrence Equations", *Journal of the Association for Computing Machinery*, Vol. 14, No. 3, (563–590), July 1967.

44. Shedler, G.S., and Lehman, M.M., "Evaluation of Redundancy in a Parallel Algorithm", *IBM Systems Journal*, Vol. 6, No. 3, (142–149), 1967.

45. Gilmore, P.A., "Structuring of Parallel Algorithms", *Journal of the Association for Computing Machinery*, Vol. 15, No. 2, (176–192), 1968.

46. Pease, M.C., "An Adaptation of the Fast Fourier Transform for Parallel Processing", *Journal of the Association for Computing Machinery*, Vol. 15, No. 2, (252–264), 1968.

47. McCluskey, E.J., *Introduction to the Theory of Switching Circuits*, McGraw-Hill, 1965.

48. Gimpel, J.F., "A Reduction Technique for Prime Implicant Tables", *Institute of Electrical and Electronic Engineers Transactions on Electronic Computers*, Vol. EC-14, No. 4, (535–541), August 1965.

49. Miller, R.E., *Switching Theory*, Vol. 1, John Wiley and Sons, New York, 1965.

50. Marimont, R.B., "Applications of Graphs and Boolean Matrices to Computer Programming", *SIAM Review*, Vol. 2, No. 4, (259–268), October 1960.

51. Bingham, H.W., Reigal, E.W., and Fisher, D.A., *Parallelism in Computer Programs and Multiprocessing Organizations*, AD 827 655, January 1968.

52. Bingham, H.W., Reigal, E.W., and Fisher, D.A., *Parallelism in Computer Programs and in Machines*, AD 667 907, April 1968.

53. Martin, D.F., and Estrin, G., "Models of Computational Systems – Cyclic to Acyclic Graph Transformations", *Institute of Electrical and Electronics Engineers Transactions on Electronic Computers*, Vol. EC-16, No. 1, (70–79), February 1967.

54. Kral, J., "One Way of Estimating Frequencies of Jumps in a Program", *Communications of the Association for Computing Machinery*, Vol. 11, No. 7, (475–480), July 1968.